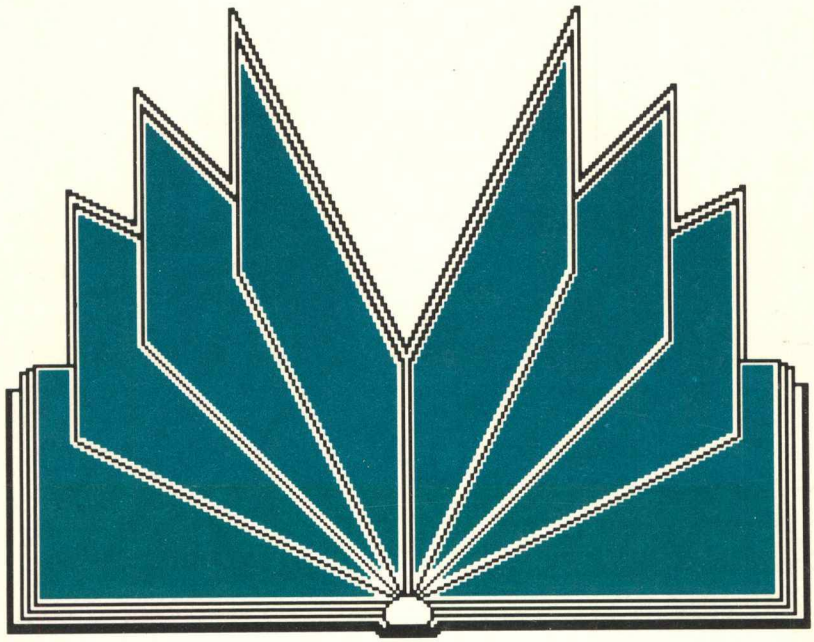
Dr. Dobb's Journal

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

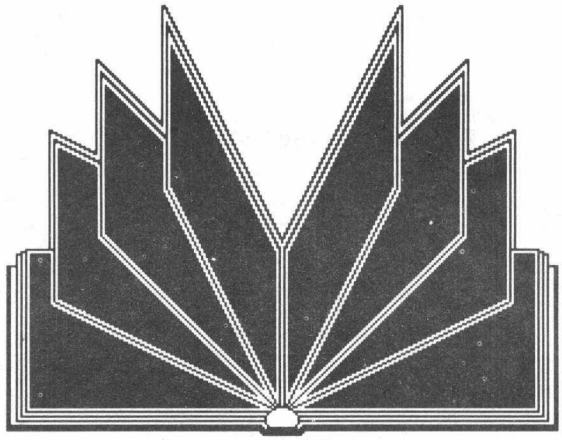


VOLUME

Dr. Dobb's Books
A Publication of M & T Publishing, Inc.

Dr. Dobb's Journal

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS



VOLUME

8

Dr. Dobb's Books
A Publication of M & T Publishing, Inc.

Acknowledgments

We have always used this space to thank our authors, programmers all, who contribute their time, knowledge, and source code for the edification of other programmers. They deserve all our gratitude: to broadcast hard-earned knowledge like seeds on the wind is extraordinary generosity. We of *Dr. Dobb's* staff are pleased to be the breeze in the metaphor.

Bound Volume Staff

Editor:
Marlin Ouverson

Book Production:
Geoffrey Brandenburg

Index:
Julie Anton

Cover Design:
Shelley Rae Doeden

Publisher's Representative:
Jane Sharninghouse

COPYRIGHT © 1985 by M&T Publishing, Inc. All rights reserved. No part of this book may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage and retrieval system, without permission in writing from the Publisher.

Printed in the United States of America

About M&T Publishing, Inc. and Dr. Dobb's Journal

M&T Publishing, Inc. was founded in 1981 as the American subsidiary of Markt & Technik A.G., the leading West German publisher of computer magazines, books and software. In addition to its active role in the international microcomputer marketplace, M&T also publishes two monthly magazines. *Business Software* was acquired in 1983, and in 1984 M&T acquired the rights to publish *Dr. Dobb's Journal* from People's Computer Company.

Since its first issue in 1976, *Dr. Dobb's Journal* has maintained a unique voice among microcomputer publications. The first issue was published to provide short-term distribution of the newly written Tiny BASIC language. Reader response was so strong that it went immediately into monthly production. *DDJ* has led the way since then by focusing on important advances in microcomputers, printing public-domain software and fostering vital reader interaction on technical matters.

Today, *Dr. Dobb's Journal* remains the prime source of information and software tools for advanced programmers. Its early issues are still in demand, and no part of the editorial content of *DDJ* has ever gone out of print. Volumes One through Eight of this series contain the entire editorial content of *DDJ*'s first eight years, and are available in bound volumes through M&T Publishing, Inc.

Through *Dr. Dobb's Journal* and its other software publications, M&T Publishing, Inc. is providing microcomputer professionals and enthusiasts with the most advanced and useful software information available.

Editor's Preface

So much code.

This was the year in which *Dr. Dobb's Journal* published the conclusion to Jim Hendrix's Small-C Version 2, one of the most popular articles ever to appear in the magazine. "Article" is the wrong word, of course, a remnant of another age of publishing. Dr. Dobb's was one of the publications that brought in the odd new era in which magazine pages were packed with articles, whole volumes, written in no human language. Small-C Version 2 is a piece of software, and a very useful one. By publishing it in source-code form, Dr. Dobb's made the C programming language available to thousands of programmers who would not otherwise have been able to learn how to use it, let alone learn how the language was constructed.

Dr. Dobb's published a lot of source code in 1983; that's what the magazine was, and is, all about. In later years some of the code would also be distributed via more modern media that print electronically and on disk. The latest version of Small-C would be one of the first pieces of disk software from Dr. Dobb's. But back in 1983, it was all software on paper, and there was indeed a lot of it, as the size of this volume attests.

In this volume you will find all the software (as well as some more conventional articles) published in *Dr. Dobb's Journal* in 1983. There's Ed Ream's RED editor, written in Small-C, and Augusta, the microcomputer implementation of a subset of the Ada language. You'll find clever algorithms for implementing co-routines, fast matrix operations, divisibility tests, B-Tree ISAM techniques. Dr. Dobb's scope has always been broad, so you'll find articles herein that include code for the 6800, 6502, 8080, Z80, 8086 and 68000 microprocessors; high-level code in Forth, C, Basic, Ada, Pascal, and some languages created for the occasion. You'll also find letters to the editor, opinionated columnists, running dialog on the craft of software development.

The point of it all is this: good programmers can learn from others' code. Despite the very real usefulness of the code published in *Dr. Dobb's Journal*, the code's greatest value is in its tutorial aspect. That's why it still makes sense to release it in so archaic a medium as this volume.

Michael Swaine, Editor
Dr. Dobb's Journal

Contents

VOLUME 8, NUMBER 75, 1983

3 Letters
4 Editorial
5 16-Bit Software Toolbox RAY DUNCAN
8 Augusta, An ADA Subset for Micros EDWARD MITCHELL
21 Xanadu: Hypertext From the Future ROGER GREGORY
24 Stone Age Computers: 6000 Years of Computing Science HANK HARRISON
29 Software Review:
"Another Look at JRT Pascal" DANIEL S. HUNT
"Rebuttal" D.E. CORTESI
31 Small-C Compiler, v.2 (continued listing) J.E. HENDRIX
46 CP/M Exchange GENE HEAD
47 Of Interest MICHAEL WIESENBERG

VOLUME 8, NUMBER 76, 1983

53 Letters
53 Editorial
55 PISTOL: A Forth-like Portably Implemented STack-Oriented Language ERNEST E. BERGMANN
57 Dobb's Ex Machina HANK HARRISON
58 16-Bit Software Toolbox RAY DUNCAN
64 Program Linkage by Coroutines — Forth to BASIC LEONARD MORGENSTERN, M.D.
70 Linking CP/M Functions to Your High-Level Program WENDY A. WEIGER
72 CP/M Exchange GENE HEAD
90 Dr. Dobb's Clinic DAVE CORTESI
92 Hardware Review: CP/M-80 Expansion Card HANK HARRISON
95 Software Review:
"REVAS Disassembler" STEVE WILLOUGHBY
96 Book Reviews
99 Of Interest MICHAEL WIESENBERG

VOLUME 8, NUMBER 77, 1983

107 Letters
111 16-Bit Software Toolbox RAY DUNCAN
114 Augusta, Part II — The Augusta P-Code Interpreter EDWARD M. MITCHELL
126 A Small-C Operating System BRIAN MCKEON
145 6809 Threaded Code: Parametrization and Transfer of Control H.T. GORDON
147 A Common-Sense Guide to Faster, Smaller BASIC ROBERT IRVING
149 A Fundamental Mistake in Compiler Design EDGAR H. FEY, JR.
151 Open Forum:
Comments on Fifth Generation Computers MICHAEL J. DOHERTY
152 CP/M Exchange GENE HEAD
155 Dr. Dobb's Clinic DAVE CORTESI
159 Of Interest MICHAEL WIESENBERG

VOLUME 8, NUMBER 78, 1983

165 Letters
165 Editorial
167 CP/M Exchange GENE HEAD
170 RECLAIM — File Reclamation Utility for Destroyed Directories WALTER V. MURPHY
176 Binary Magic Numbers EDWIN E. FREED
184 An 8080 fig-FORTH Directory and File System ALFRED J. MONROE
194 SAY" Forth Votrax Driver" C. KEVIN MCCABE
198 A TRS-80 to 8080 Translator ANTHONY T. SCARPELLI
212 Dr. Dobb's Clinic DAVE CORTESI
214 16-Bit Software Toolbox RAY DUNCAN
218 Software Review:
"Unica and XM-80" MICHAEL FAVITTA
221 Of Interest MICHAEL WIESENBERG

VOLUME 8, NUMBER 79, 1983

227 Letters
227 Editorial
229 Dr. Dobb's Clinic DAVE CORTESI
231 Augusta, Part III — The Augusta Compiler EDWARD MITCHELL
247 A Fast Circle Routine DANIEL L. LEE
251 Enhancing the C Screen Editor ALAN D. HOWARD
268 Shifts and Rotations on the Z80 RON GOODMAN
270 The SBC, TSX and TXS Instructions on the 6502 and 6800 B.T.G. TAN
272 Software Reviews:
"ZAS Z8000 Development Package" TERRY R. DETTMANN
"6809 Cross Assembler" TERRY DETTMANN
273 Book Reviews
275 CP/M Exchange ROBERT BLUM
278 16-Bit Software Toolbox RAY DUNCAN
280 Of Interest MICHAEL WIESENBERG

VOLUME 8, NUMBER 80, 1983

285 Dr. Dobb's Clinic DAVE CORTESI
287 Fast Divisibility Algorithms H.T. GORDON
289 B-Tree ISAM Concepts CHRIS DEPPE and ALAN BARTHOLOMEW
293 CP/M BDOS and BIOS Calls for C TERJE BOLSTAD
298 Printing Graphics on the IBM PC DAN DAETWYLER
304 Game of LIFE on the IBM PC SIMSON GARFINKEL
311 Series Expansion in Forth WENDALL C. GATES
315 Fast Matrix Operations in Forth, Part I STEVEN A. RUZINSKY
318 Yes, You Can Trace Through BDOS DO-WHILE JONES
323 Julian Dates for Microcomputers GORDON KING
328 16-Bit Software Toolbox RAY DUNCAN
334 CP/M Exchange ROBERT BLUM
336 Of Interest MICHAEL WIESENBERG

VOLUME 8, NUMBER 81, 1983

345 Letters
345 Editorial
348 Augusta, Part IV — The Augusta Compiler (continued) EDWARD MITCHELL
362 RED — A Better Screen Editor, Part I EDWARD K. REAM
380 Anatomy of a Digital Vector and Curve Generator RICHARD SHIFFMAN
383 Fast Matrix Operations in Forth, Part II STEVEN A. RUZINSKY
392 Dr. Dobb's Clinic DAVE CORTESI
396 CP/M Exchange ROBERT BLUM
400 16-Bit Software Toolbox RAY DUNCAN
406 Of Interest MICHAEL WIESENBERG

VOLUME 8, NUMBER 82, 1983

413 Letters
413 Editorial
415 Dr. Dobb's Clinic DAVE CORTESI
422 Serial-to-Parallel — A Flexible Utility Box ALAN D. WILCOX
428 McWORDER — A Tiny Text Editor WILLIAM A. MCWORTER
433 Fast Matrix Operations in Forth, Part III STEVEN A. RUZINSKY
441 And Still More Fifth Generation Computers! RICHARD GRIGONIS
446 RED — A Better C Screen Editor, Part II EDWARD K. REAM
470 CP/M Exchange ROBERT BLUM
478 Software Review:
"ACTxx Cross Assemblers" STEVE NEWBERRY
480 Book Reviews
483 Of Interest MICHAEL WIESENBERG

VOLUME 8, NUMBER 83, 1983

489 Editorial
490 Letters
492 Dr. Dobb's Clinic DAVE CORTESI
495 Forth and the Motorola 68000 JOE BARNHART
502 A 68000 Forth Assembler MICHAEL A. PERRY
510 Nondeterministic Control Words in Forth L.L. ODETTE
516 GO in Forth C.H. TING
523 SEND and RCV: A Forth Implementation of the XMODEM Protocol ROBERT TAYLOR III
529 Precompiled Forth Modules N. SOLNTSEFF
533 Signed Integer Division ROBERT L. SMITH
535 When a Page is Not a Page: Forth-83 and Vocabularies GEORGE WILLIAM SHAW II
538 Some Forth Coding Standards EDWARD WISCHMEYER
542 The Forth Sort MARK I. MANNING
546 CP/M Exchange ROBERT BLUM
551 16-Bit Software Toolbox RAY DUNCAN
557 Of Interest MICHAEL WIESENBERG

VOLUME 8, NUMBER 84, 1983

563 Letters
564 Editorial
565 Dr. Dobb's Clinic DAVE CORTESI
567 C/Unix Programmer's Notebook ANTHONY SKJELLUM
570 UNIX to CP/M Floppy Disk File Conversion W.E. WILSON
583 A Small-C Help Facility JOHN STANEFF
600 Attaching a Winchester Hard Disk to the S-100 Bus OSCAR GOLDMAN
608 Using Epson Bit-Plot Graphics DON GAY
618 16-Bit Software Toolbox RAY DUNCAN
627 CP/M Exchange ROBERT BLUM
631 Of Interest MICHAEL WIESENBERG

VOLUME 8, NUMBER 85, 1983

637 Letters
638 Editorial
639 Dr. Dobb's Clinic DAVE CORTESI
646 A Kernel for the MC 68000 STEVE PASSE
662 A DML Parser DAVID HARRY
688 Toward a More Writable Forth Syntax HARVEY GLASS
692 Simple Graphics for Printer WIL BADEN
699 Software Reviews:
"WP6502 v1.3a" JAMES CRISCIONE
"DATA CAPTURE/pc" CHUCK CRAYNE
"/HELP" DIANE CRAYNE
700 Book Reviews
702 CP/M Exchange ROBERT BLUM
707 16-Bit Software Toolbox RAY DUNCAN
712 Of Interest MICHAEL WIESENBERG

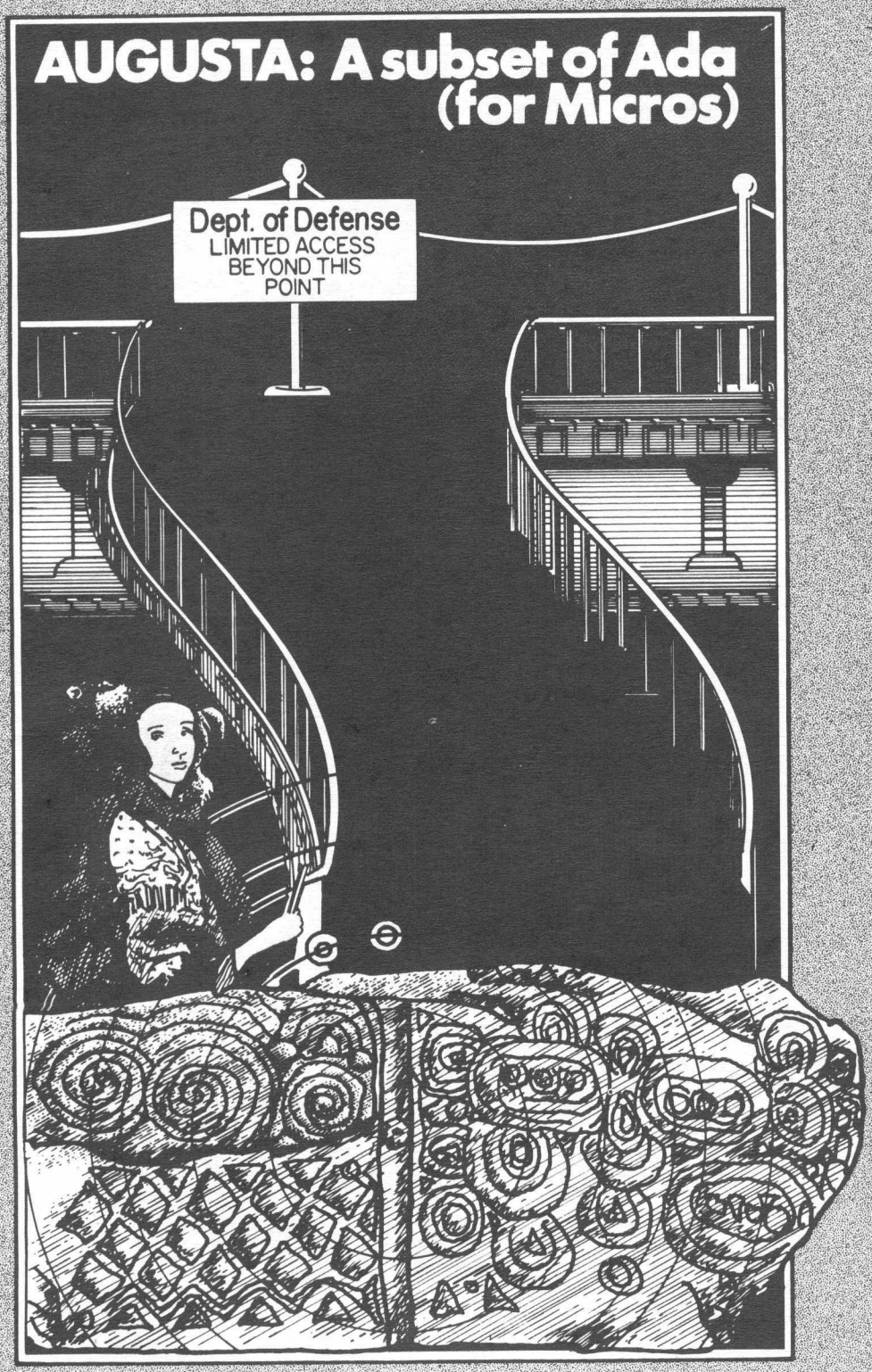
VOLUME 8, NUMBER 86, 1983

717 Editorial
717 Letters
719 Dr. Dobb's Clinic DAVE CORTESI
720 C/Unix Programmer's Notebook ANTHONY SKJELLUM
723 Faster Circles for Apples MYRON L. PULIER
733 Cursor Control for Dumb Terminals IAN ASHDOWN
738 Dysan's Digital Diagnostic Diskette LOREN AMELANG
768 Building a Programmable Frequency Synthesizer MICHAEL L. SIMON
774 CP/M Exchange ROBERT BLUM
781 16-Bit Software Toolbox RAY DUNCAN
788 Software Reviews:
"TRANSFORM" MICHAEL FAVITTA
"COMMX-PAC" GENE HEAD
791 Book Reviews
794 Of Interest MICHAEL WIESENBERG

#75 January 1983 $2.50 ($3.00 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems



XANADU

Stone Age Computers

Small-C Compiler v.2

(Continued Listing)

Dr. Dobb's Journal
For Users of Small Computer Systems

January 1984
Volume 9, Issue 1

CONTENTS

ARTICLES

24 NBASIC: A Structured Preprocessor for MBASIC
by Namir Clement Shammas

This preprocessor provides new commands and features for MBASIC and BASICA programs, including alphanumeric labels, as well as REPEAT and CASE structures. This eliminates the need to use line numbers in branching operations, allowing more structured programs to be written.

36 A Simple Window Package
by Edward Mitchell

This program, written in Augusta (the tiny language introduced last year in *DDJ*), shows how to produce simple windows on your computer display. While written to work on the Osborne computer, the author provides tips on how to modify the package for the IBM PC and other computers.

44 Forth to PC-DOS Interface
by David Cornell

In bringing up Forth on his IBM PC, the author needed a way for Forth to access the resources of MSDOS and the PC's machine language routines. The solution was two special vocabularies for transferring parameters to and from registers during function calls.

60 Sorted Diskette Directory for the IBM PC
by Dan Daetwyler

This handy utility for your IBM PC sorts and formats directory data, and then prints the directory to fit inside the diskette envelope. The program runs under PC-DOS versions 1.x or 2.0, and should operate on any normal configuration.

76 Emulate WordStar on TOPS-20
by Steven Fisher

If you, like the author, are faced with working on many different machines, each with its own editor commands, you too may long for a single command set. This article shows how to make a popular video editor for DEC's TOPS-20 system capable of accepting WordStar commands.

DEPARTMENTS

6 Editorial

8 Letters

18 Dr. Dobb's Clinic

More on optimizing compilers and some problems under CP/M Plus with PAUSE, as well as two public domain programs.

78 CP/M Exchange

A bit on directions for the Exchange and the industry, the PIP mystery device contest, and mapping your I/O ports for serial interfacing.

83 16-Bit Software Toolbox

Feedback on Floating Point Benchmarks, More on DEC Professionals, Linking Assembler and Microsoft BASIC, and an improved CLINK utility.

88 Book Reviews

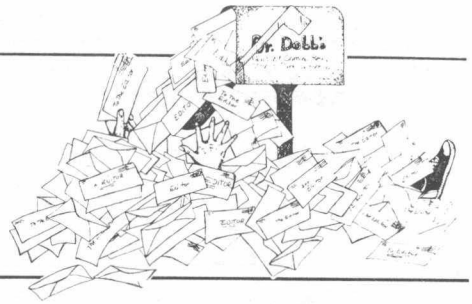
PC DOS: Using the IBM PC Operating System, Soul of CP/M, Third Caltech Conference on Very Large Scale Integration

90 Software Reviews

SuperScripsit, COMPAC, MCDISPLAY, SMAL/80

104 Of Interest

LETTERS



Split Decision

Dear Mr. Harrison:

I have a software "need" I would also like to lay out to you and potential readers. There appear to be inherent advantages to using the CRT as something more than a *tunnel* into one corner of the computer's "output buffer." In fact, there seems no reason why programmers would not reap immense benefit from the flexibility to define alternative "windows" into various portions of memory and disk storage. Larger computers and operating systems and related software are presently making available (mostly through graphic software innovations designed a decade ago but only now generally implemented) multiple-window displays. An excellent model for this is Smalltalk and Xerox Altos' and Star's multi-window environment. The Corvus Concept (limited in other respects) has had its word processing software designed for multiple views of different (or same) documents. Several software items (i.e., word processors like Mince and its copy Perfect Writer) allow split-screen display as does a file comparator utility by Leor Zolman. But these are all partial implementations outside of a larger operating system environment within which the user should have the ability to *choose* single or multiple displays. Such displays would enable the user to view the (static) contents of files while writing modules from them into a currently edited program file on a second window. With even split screen, file comparisons or the ability to monitor program compilation while editing other files or even running a program could be accomplished. To keep it simple for a CP/M environment, in a split-screen environment, one screen could always be inert while the other is "active," and still provide significant information enhancement.

I am only a beginner in this age of micros and software, and so still tremble at the thought of re-tooling the BIOS or other innards of an OS or display interface, yet feel a bit puzzled at why others have not seen merit in this approach, at least to offer at the OS level a choice of whether or not to opt for single, full-screen display or for two or more screens (partitions of the full screen) within which different files or operations can be loaded. If Mince can do it, why can't the OS? Have you any leads as to programmers who have gone after this generalized kind of utility?

I find *DDJ* most welcome in terms of its serious and technical approach to micros and software development. I shall stay tuned.

Sincerely,
Andrew N. White
2211-A Mohala Way
Honolulu, Hawaii 96822

Sole-Saving Subscription

Dear Doctor Dobb,

I'm sick of walking to the newsstand to find that your magazine has been sold out. Please enter me on your subscription list. Enclosed is a money order for $25.00.

Thank you for an enjoyable and informative magazine.

Sincerely yours,
William C. DuPree
511 Clyde, Apt. 30
Calumet City, IL 60409

We are working on this. Perhaps we should merge with High Times. *I didn't know there were any newsstands in Calumet City carrying* DDJ, *but take heed, the subscription route is the "path of choice." – Ed.*

A Dis-Contented Reader

Dear Hank:

When *DDJ* and *BYTE* (and, to a lesser degree, *Kilobaud*) started, I found most of the content informative and stimulating. Now, for the last two years or so, *BYTE* has had (on the average) only one article of sufficient technical content and level to justify the time to read it, per month. *DDJ* (alas!) is, in my opinion, down to one per *year*.

Thanks for being at the front of the micro revolution. I'm really sorry your editorial policy has not attracted more good content that *educates* rather than gives low-level listings of yet another assembler or monitor. I have nothing against listings – they are often necessary – but they are not very informative *as such*. Perhaps tables and figures focused on program variables, algorithms, and flow charts would be more helpful.

Tom Pittman
San Jose, CA

Computer as Salesperson?

Dear Hank:

The following idea might, for all I know, already be in application in the U.S. networks. (Here in New Zealand a government sales tax of 40% has strangled growth of personal computers.)

Commercial networks could be used for a "try before you buy" preview of software. Because of slow baud rates, interactive responses would be limited; that is, graphics games would be demonstrated as snapshots of how the screen looks. Data manipulation programs like accounting programs and even editors could be demonstrated – albeit slowly – in real time. There would be some sort of hourly charge for use. Thus, you could try a program for a while without necessarily buying it.

Only after you decided to purchase the software, and gave the system your order code and password, would the program be either down-loaded into your computer or else sent on magnetic media via post. To the extent that this would make acquiring software easier than going to a friend's house, it would reduce software piracy.

There would not be any real problem in restricting these demonstrations to any given computer, and in fact, this mass-marketing approach could greatly increase software availability for people with oddball machines spread thinly around the population base.

The early concept of using a network to access a big, powerful computer is out of date, as personal computers become more powerful. My proposal is that the large computer act as a salesman.

Cordially yours,
Jay D. Mann
330 Centaurus Road
Christchurch 2
New Zealand

Why Pascal at All?

Dear Doctor Dobb,

The cheapie JRT system is really useless because it doesn't know how to do arithmetic! There are at least three problems with its "real" number arithmetic in addition to its compiler bugs.

When finite-register machines try to approximate the real number continuum, they tend to fail as the numbers approach zero. When the absolute value of the number is less than the minimum allowed in the representation, the stored result should be identically equal to zero. Perhaps the implementers of JRT Pascal never learned this in school.

There is one more arithmetic bug that makes the system useless for account-

(Continued on page 12)

EDITORIAL

High-Tech Writer

Hello: My name is Hank Harrison. As a long-time reader and now the new Editor of *Dr. Dobb's Journal* I'm convinced that *Dr. Dobb's* is the finest small publication in the world for the microcomputer community. It is good, not because it is engorged with advertisements, not because it appeals to the hobbyist, or is mistake free, but because, since 1976, *Dr. Dobb's* has been read and respected by the real pioneers and inventors of our trade . . . the *Gyro Gearloose* computerists.

For many issues past, Marlin Ouverson has been sitting in this chair and running *Dr. Dobb's* well. I have no intention of changing his good works. I think, however, that since I grew up in a tough town, Hayward, CA, and have just returned from two years in Dublin, Ireland, lecturing in Industrial Psychology at Trinity College and looking for Celtic artifacts, I will have to be slightly different – not radically different, but, well, just Plain Vanilla me. For the past two decades I have earned a sporadic living as a tech writer.

* * *

Recently *Dr. Dobb's* lost a greatly needed bunch of money by taking an ethical stand with an advertiser. Two of our top experts said the product didn't work and wasn't worth the very low price advertised. The owner of the company wrote back and claimed he was disgusted, then another letter appeared in another well-respected weekly journal confirming our research, then we received other letters, quite independently and without solicitation, also claiming the product is awkward and unreliable, and wondering how it could have received such a good rating from the well-respected weekly news magazine. The advertiser who claims to have been disgusted, asserts that three other magazines of great reknown paid homage to his product and why shouldn't we?

About a week later, we received a very well written article claiming that the product is flawed but fabulous. This long piece was written, not by a homebrew person, but by a professional with a lot of experience with similar products. He claims the product remains unfriendly, but is no more or less unfriendly than other similar products costing ten times as much. I checked on this claim, and reread the two reviews in the other magazines. Neither was a rave as the advertiser had claimed, but they were still positive reviews. I then went to our old files and located certain press releases. Guess what? Many elements of the PR material appeared in the two favorable reviews. Then the entire issue took on a new dimension. The next day, one of our own readers wrote in to tell us that she hasn't had a chance to use the product because it just never arrived in the mail.

I'd like to mention the fact that *DDJ* takes the time to "handle" the software and hardware we feature and we try to let the reader draw his or her own conclusions.

* * *

In going over the past issues of *Dr. Dobb's*, I noticed an amazing dearth of women contributors. I realize that this is traceable to the education system and an industry-wide bias. I also realize that women in the systems development area may be too busy to send in articles to *DDJ*, but I can safely say we are looking for articles by women, simply to balance things out and also to honor an old obligation of mine – I think it would be good for the industry.

Another point must also be considered, having to do with a controversial suit being brought against a small game manufacturer for releasing an odious little game wherein General Custer, true to his reputation, rapes an Indian maiden until shot with an arrow. This offense is not as pornographic as Women Against Pornography depict it, nor is it a simple affront to women, per se, since all acts of rape are an affront to both genders and a few in between. This Wild West maze game is a distortion of history and an affront to Native Americans because they won the battle with Custer the first time. The massacre isn't reflected in the game. This kind of tripe is an assault on the ethics of the entire microcomputer industry. Is Willy Loman moving in?

How is it possible that some really ugly people were allowed to produce such a game in the first place? Is the First Amendment at risk here? You can't tell me someone with authority didn't see this one coming. Have we slipped this far? Who wrote the program? How many people saw the game under development and said nothing? "Hmmm, things are getting curiouser and curiouser," said Alice (in Wonderland).

* * *

Last week I received a call from the most famous board game manufacturer in existence. They were looking for a back issue of *Dr. Dobb's* that detailed their game, in software. Apparently they want to computerize the game for international release. They stand to make a fortune. The guy that blissfully wrote that old program won't get a penny. He put it in public domain and listed the code in *Dr. Dobb's*.

We at *Dobb's* have long been thought of as consumer advocates. We will therefore be running more *Dobb's Ex Machina* as the time goes on. In that context, we will be emphasizing the legal ramifications of the public domain software issue. So if you're looking for changes, look for them in terms of an intensification of our traditional ethics. Remember, Willy Loman is lurking in the wings, and he can't program, and if you don't know who Willy Loman is, ask the ghost of Marilyn Monroe.

Hank Harrison
Senior Editor
Dr. Dobb's Journal

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

Recent Software, Briefly Reviewed

DOSCPM, by Adaptive Computer Systems, P. O. Box 280B, Wheeling, IL 60090. $50. An efficient and compact utility to convert PC-DOS disk files to CP/M-86 format. It's easy to use, very reliable, and has excellent error handling. The six pages of documentation fit nicely into the CP/M-86 operator's manual. It has two minor drawbacks: it only copies files one direction (does not convert CP/M-86 files to PC-DOS), and it can't handle double-sided PC-DOS disks. Note that although PC-DOS programs can be transferred onto a CP/M-86 diskette, they cannot be executed because of differences in the load image and operating system call conventions.

dBASE II version 2.3B for the IBM Personal Computer, by Ashton-Tate. Operationally identical to the CP/M-80 version, but it requires 96 Kbytes of memory for no obvious reason. The main DBASE.COM file, the overlays, and the message file take up about 64 Kbytes of disk space, leaving plenty of room on a double-sided disk for other command and data files. The editing control codes are essentially the same as in the 8080/Z-80 version; the PC's arrow keys are also active but no use was made of the other special function keys. In spite of its relatively high price, this is one of the few packages I have purchased that is really worth every penny. You will never cease to find new uses for this powerful general-purpose data base manager.

Screen Print Program for the IBM PC, by M. A. P. Systems, Inc., 1120 NASDA Road One, Suite 444, Houston, TX 77058. $44.95. A set of utility programs which transfer a text or graphic image from the video display to an Epson printer equipped with the Graftrax option. Distribution disk includes low- and high-speed versions of three different dump formats: normal-size/upright, normal-size/rotated, and double-size/upright. After one of the dump programs is invoked, it becomes resident in RAM and transparent to the operation of other application software. A screen dump can be triggered at any time by keying "Shift-Prtsc." This is a very smart program: it handles true bit-mapped graphics or the text mode "business graphics" equally well, and also provides shading when reproducing a color graphics image. It will probably be most useful to people who wish to put screen examples in their software manuals. Very nicely implemented.

Concurrent CP/M-86 for the IBM PC, by Digital Research, Inc. A real-time, multi-tasking operating system that leaves PC-DOS light-years behind. I plan to present a detailed review of this ground-breaking product in the next issue of *DDJ* (unless Dave Cortesi beats me to it). For now, it suffices to say that Concurrent CP/M-86 will make possible applications of unprecedented power in a desk-top computer. However, it is not for the user who is faint-of-heart or on a limited budget – aside from the $350 cost of the software, the PC must contain 192 Kbytes of RAM to even boot up the operating system, and Digital Research recommends a minimum of 256 Kbytes.

FASTPAK 8087 support subroutine libraries for the IBM Personal Computer, from Microware, P. O. Box 79, Kingston, MA 02364. $375.00. Includes an 8087 numeric co-processor chip and your choice of run-time libraries for the IBM Pascal, BASIC, or FORTRAN compilers. A package of subroutines and macros for the assembly language programmer is also available.

Coping with PC-DOS

Isaac Davidian writes: "The comments in the November 1982 column about wiping out a diskette directory under PC-DOS if the incorrect disk is installed struck home with me. I had this problem when making copies on a single drive by swapping disks, and accidentally inserted the wrong disk.

"Unfortunately, I had no backup and the disk contained months of work on BASIC programs. I did, however, find a way to recover the files, and if anyone has this problem I would suggest setting the disk aside in the event they wish to try recovery.

"Briefly what I did was to create a BASIC program file (with line numbers in the 60000 range) on a new diskette. This file should occupy one of the larger continuous blocks on the diskette and it may require saving the file more than once under different names.

"The way to check this is to use DEBUG to read the diskette. Make note of the file location on the diskette and note the header data, including file names.

"You must then find, on your damaged diskette, each block which contains a portion of your program. You then read the block from the damaged diskette and write it to the new diskette specifying the sectors in the dummy file you created. Before you write to the new diskette, however, you should change the header information to conform to the dummy file.

"After this, you can exit DEBUG, use BASIC to load the dummy file, and then save it under a different name. The process must be repeated for each group of sectors on the damaged diskette which contains the file you are recovering.

"Since program lines do not begin and end in sequence with the blocks of data on the diskette, you may lose portions of your program, or you may have to wipe out a partial line at the beginning of succeeding blocks that are written to the new file."

Readers who are interested in helping Isaac refine this technique further can contact him at 5150 N. Sixth, Suite 150, Fresno, CA 93710.

Light Pens on IBM PC

The color/graphics video board on the IBM Personal Computer includes a light pen interface, and some light pen control words are included in the PC's BASIC interpreter. After years of jealousy seeing light-pen-oriented, menu-driven applications on IBM mainframe systems, I bought the first available light pen, plugged it in, and prepared to have a good time. I am sad to report that, although the light pen interface will be adequate for picking choices from menus, some corners were cut that will make it unusable for detailed design work. The resolution of the controller on the X axis is limited to that of text mode, even when the video board is operating in graphics mode. For example, in high-resolution graphics mode, pointing the light pen at all of the points (x,y)=(0,0) through (7,0) will return the value (0,0), the points (8,0) through (15,0) all return (8,0), and so forth. Similarly, on the Y axis the driver only returns values that are multiples of two.

Light pens for the PC are manufactured by Symtec Inc., 15933 W. 8 Mile Road, Detroit, MI 48235, and FTG Data Systems Inc., P. O. Box 615-C, Stanton, CA 90680.

Trigonometric Algorithms

Many new microcomputer programmers regard the math function libraries

of the commonly available interpreters and compilers as "magical," feeling that their internal operation must be too complicated to understand. Not so! The people who implemented most of these packages were not smarter than you – they just have access to the right information. There is an extensive body of literature, worked out and refined by mathematicians over a period of many years, that concerns itself with the efficient calculation of advanced numeric functions.

Many of the functions of interest to us, such as exponentiation, sine, and cosine, can be expressed as the sum of an infinite series:

$$f(x) = \sum_{n=0}^{m} a_n x^n$$

The appropriate coefficients for each function are available in many books of mathematical tables. Although these series are very exact, they tend to converge slowly and therefore require a great deal of computation time when implemented in a program. There are ways of "economizing" these series so that they converge more quickly and produce results within a predictable range of error; two of these methods are called Chebyshev approximations and rational polynomial approximations.

For example, the full series expansion of e^x is:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} \cdots$$

By techniques of Chebyshev approximation, the first seven terms of the series as shown above can be reduced to:

$$e^x \cong 0.9946 + 0.9971x + 0.5430x^2 + 0.1776x^3$$

which has about the same percentage error over the interval $0 \leqslant x \leqslant 1$ and requires considerably fewer multiplication operations. Such approximations are at the heart of all compiler run-time math libraries and are also used in pocket calculators and the microcode of numeric processors such as the Intel 8087 or AMD 9511.

For the sake of giving you an inkling how floating-point packages work, I will now present a very simplified technique to calculate sine, cosine, and tangent of a given angle directly.

A Maclaurin series expansion of sin(x) is:

$$\sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} \cdots$$

where x is in radians (2*PI radians = 360 degrees). This can be economized by a rational polynomial approximation to:

$$\sin(x) \cong \frac{x - \frac{14}{120}x^3}{1 + \frac{1}{20}x^2}$$

This formula is accurate to about 1 part in 10,000 over the interval $0 \leqslant x \leqslant \pi/4$, a degree of precision which is adequate for typical microcomputer applications. For x greater than $\pi/2$, the error increases substantially and the function becomes essentially unusable for our purposes.

So now we have a function that will give us sin(x), but only for the first eighth of a circle (a portion commonly called an octant). How do we generalize this so that it can be used for any angle? Luckily, the trigonometric functions have a recurring, predictable nature and there is a set of "identities" that relates the values of sine and cosine in all of the octants to their values in the first octant (see Table 1). Wherever cosine appears in the table, you can substitute:

$$\cos(x) = \sqrt{1 - \sin^2(x)}$$

by virtue of the trig identity:

$$\sin^2(x) + \cos^2(x) = 1$$

We are now in a position to calculate the sine or cosine of any angle directly, and can also obtain tangent(x) by the relation:

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

In summary, we can derive the values of sine or cosine for any angle simply by determining its octant, looking up the relationship for the relative angle within that octant to sin(x) in a simple table, then applying the approximation function. Tangent is obtained by calculating both sine and cosine and then finding their quotient.

If you were involved in a project requiring high-powered math, and did not want to waste months "reinventing the wheel," you could buy the source code from United States Software Corporation, 5470 NW Innisbrook Place, Portland, OR 97229.

DDJ

Let R = absolute value of angle MOD PI/4

Let S = –1 or 1, depending on the sign of the angle

octant	sine	cosine
0	S * sin(R)	cos(R)
1	S * cos(PI/4 – R)	sin(PI/4 – R)
2	S * cos(R)	–1 * sin(R)
3	S * sin(PI/4 – R)	–1 * cos(PI/4 – R)
4	–S * sin(R)	–1 * cos(R)
5	–S * cos(PI/4 – R)	–1 * sin(PI/4 – R)
6	–S * cos(R)	sin(R)
7	–S * sin(PI/4 – R)	cos(PI/4 – R)

Table 1.

The relationship of sine and cosine of an angle in each octant. (Octants are numbered counter-clockwise starting at 0 radians.)

Letters *(Continued from page 6)*

ing applications, as well. In the statement: c:=a * b; if any of the variables is "real" and b = 0, then c will always equal a. One interesting result of this bug is that the system LOG procedure insists that LOG(1.0) = 2.

JRT Pascal is the cheapest way to find out for yourself that Pascal is a miserable language for serious applications, because too much in the way of input/output control has been left out and too many redundant requirements for source code were put in. Pascal was designed for the convenience of the compiler writer, rather than that of the programmer. To cap it off, any Pascal includes what is probably the worst programming practice known to modern science: running once through a loop before making any tests [REPEAT . . . UNTIL].

One wonders why anyone but a "computer science" major would go through the effort?

Yours very truly,
Murray L. Lesser
2474 Hunter Brook Road
Yorktown Heights, NY 10598

Late for the Train

The following letter was sent to us as a copy of one sent to JRT Systems, Inc.

Gentlemen:

On September 5 I ordered your Pascal program for my North Star computer. I inquired about my order on September 25 since I did not receive any shipping notice or order acknowledgement.

On October 12, I had still heard nothing about my order, so I telephoned. The woman at your office assured me that my order would be shipped by October 15.

My latest VISA statement showed that I was billed $29.95 by your firm on September 30. My order was obviously not shipped then, nor have I yet received it.

I have ordered several software packages by mail in the past and never had this poor service. Since I responded to your ad in *Dr. Dobb's Journal*, I am letting them know my experience with your company.

If I do not receive your program or a full refund by November 16, I am registering a complaint with the U.S. Postal authorities.

Yours sincerely,
J. H. Peters
929 Robin Drive
Seaford, Delaware 19973

You ain't seen nothin' yet! See this month's Software Review for still more on JRT Pascal from two who have *received their copies. – Ed.* DDJ

AUGUSTA
An ADA Subset for Micros

by Edward Mitchell

Editor's note: Since the Department of Defense, DoD, often referred to as the Directors of Death, has not seen fit to franchise any "official" subset of Ada, we will all be seeing creative adaptations of it as time goes on.

Augusta is the first of these ADAoids to run on a portable micro, as it was tested effectively on the Osborne. It also runs on any Z80-based system and will soon be available in MS-DOS. Augusta is a copyrighted product soon to be released as a package to retail for under $100.00, tax and license included. This article is in fact the first serial of four installments to be run every other month in Dr. Dobb's. *The four segments comprise a book which should be available in 1984. The lag period between serializations is designed to allow feedback which both* DDJ *and the author encourage. Ed Mitchell is twenty-four years old.*

The Augusta[1] compiler introduced in this four-part series implements a subset of the U.S. Department of Defense Ada[2] programming language. The series describes the Augusta language, the "p-code" interpreter, the recursive descent parser and the Augusta compiler.

Ada is a strongly typed and structured language with influences from Pascal, Algol and SIMULA. (See reference 1.) "Ada" is a registered trademark of the U.S. DoD and only compilers that conform to the formal Ada definition may use the name "Ada." For that reason, this subset implementation is named "Augusta." Augusta is not a pure subset implementation of Ada and does not purport to be an Ada Compiler either now or in the future.

Augusta, like Ada, was named for Augusta Ada Byron, the Countess of Lovelace and one of the first programmers, having worked with Charles Babbage's mechanical computing engines during the mid-19th century.

Augusta is written in Microsoft BASIC-80, originally on an Osborne 1 computer. The compiler will run on other microcomputers with little or no translation, including the IBM Personal Computer, the Apple II with Z80 Softcard, and the Radio Shack Model II. The compiler generates code for a hypothetical "stack" machine. This stack machine executes "pseudo-code" or "p-code" instructions and is simulated by a p-code interpreter program.

When Augusta is run as an interpretive BASIC program, it compiles at about 40 to 50 lines per minute. Once Augusta is compiled with the Microsoft BASCOM compiler, it compiles Augusta source code at well over 300 lines per minute. The BASIC version of the p-code interpreter (described in Part 2) was used during system development as a test bed. A CP/M Z80 version of the interpreter executes p-code programs 10% to 80% faster than CBASIC programs, depending on the benchmark test.

When run using interpretive BASIC, the compiler requires about 17K bytes of memory after BASIC-80 has been loaded, plus about 4K for storage of Augusta's internal variables. On the Osborne 1, that leaves about 8K bytes, which is used almost exclusively for symbol table space at 17 bytes per symbol.

The compiled code is written to a disk file and is loaded into virtual memory at run time by the p-machine simulator. Since the p-machine is designed specifically to execute Augusta programs, the compiled version of a program is extremely compact.

Part 1 of this four-part series describes the Ada subset that forms the Augusta language and provides examples of Augusta programs. Part 2 is a detailed description of stack machines and the Augusta p-machine in particular. Part 3 is a general introduction to recursive descent compilers. Part 3 also presents the first half of the compiler source listing with the second half appearing in Part 4. Part 4 concludes the series by describing the compiler's internal operation.

Copyright © 1983 by Edward Mitchell, Graduate, Computer and Information Sciences, University of California, Irvine.

Defining the Language

Before the compiler can be written, we must precisely define the syntax of the Augusta language. Obviously, much of the full Ada definition must be omitted. For example, full Ada provides for the concurrent execution of several processes. But multiprocessing requires an operating system to manage system processes and resources. Hence, multiple processes would be difficult to fit into a small implementation.

Similarly, "packages" or program libraries entail substantial overhead inside the compiler. Other features must be left out because the compiler is written in BASIC and their implementation would be difficult.

Based on these and other restrictions, a suitable Ada subset was chosen. Ideally, the subset will be sufficient to rewrite Augusta in Augusta. Further, programs written in Augusta should be upwardly compatible to Ada with only minor modifications.

Two extensions were made to the Ada subset. In procedure calls, the "@" symbol is placed before variable identifiers when passing parameters by *reference* instead of by *value*. The use of the "@" symbol is described in greater detail later in this article. In Augusta, strings are represented internally more like UCSD-style Pascal strings than like Ada strings.

The syntax diagrams in Figure 1 are a concise, graphic description of the language. The diagrams show the sequence of language symbols or "syntax" that make up a valid Augusta program. Items within circles are language keywords like PROCEDURE", "IF", "THEN", and so on, or "tokens" such as "+", "(", or ">=". Items within rectangles are the names of descriptions shown elsewhere in Figure 1.

Syntax diagrams show the language in an outline-like form. For example, a "COMPILATION" is an optional "PRAGMA" (a special type of program comment) followed by the keyword "PROCEDURE" followed by a "PROC. HEADER" followed by a period. "PROC. HEADER," defined below "COMPILATION," describes the syntax for a procedure.

An Overview of the Language

Augusta programs *look* a lot like Pascal. But there are important differences, particularly when compared to the "tiny" versions of Pascal. Listing 1 shows a sample program listing produced by the Augusta compiler. Note that program "comments" are marked by placing a double hyphen "--" before the comment text. The four columns to the left of the source text represent (1) the line number in the source text, (2) the proce-

dure number, (3) the number of bytes of code generated up to that line for the current procedure, and (4) the offset from the frame mark for the variable definitions.

Augusta programs are typically split into several procedure, function, and variable definitions. Variable and procedure names consist of alphabetic and numeric characters and the underscore "_". For example, all of the following may be Augusta identifiers,

```
X
ALPHA
STORE19
NEW_LINE
COST_FUNCTION
```

Identifiers may be any length (up to 255 characters), but only the first 8 characters are significant. For example,

```
ALPHABETA
```

and

```
ALPHABETIC
```

refer to the same identifier because the first 8 characters are the same. (The underscore character "_" does not count as a significant part of the identifier name: "X_Y" is equivalent to "XY" since the "_" is not stored in the symbol table.)

Augusta provides four data types: INTEGER, CHARACTER, STRING, and BOOLEAN, plus constant definitions. (Four special types are also provided and are discussed later.) In addition, one dimensional arrays of each type are permitted (see Listing 1 for examples of variable declarations).

INTEGER variables hold numbers in the range -32768 to 32767. For example,

```
I, J, K:INTEGER;
```

defines the symbols "I", "J", and "K" as integer variables. They might be assigned values as follows:

```
I:=35;
J:=I*I;     – J equals I squared
K:=J+J;     – K equals the sum of I
              squared
```

BOOLEAN variables hold a TRUE or FALSE value and are often used as "flag" variables. For example, a loop is to be repeated until a certain condition tests as false. In Augusta this might be written as,

```
MOREINPUT := TRUE;
WHILE MOREINPUT
LOOP
   – process input
END LOOP;
```

The code that processes the input sets MOREINPUT to FALSE when the loop is completed. Augusta treats any non-zero value as TRUE and a zero value as FALSE.

CHARACTER variables hold a single

Listing 1.

```
2   0   0   0 PRAGMA CRT(ON); -- LIST COMPILATION TO SCREEN
3   0   0   0
4   0   0   0 -- Augusta DEMONSTRATION PROGRAM
5   0   0   0 -- Written by Edward Mitchell, 07/29/82
6   0   0   0
7   0   0   0 PROCEDURE NUMGUESS IS
8   1   0 -14   LOWBND  : INTEGER;        -- The lower bound of the guessing interval
9   1   0   2   HIBND   : INTEGER;        -- The upper bound of the interval
10  1   0   4   AGAIN   : BOOLEAN;        -- Set to FALSE if the user doesn't want more games
11  1   0   6   CH      : CHARACTER;      -- Dummy variable for input
12  1   0   8   S       : STRING; -- Dummy string for input
13  1   0  89   NGUESSES: INTEGER;        -- # of guesses made by computer
14  1   0  91
15  1   0  91 FUNCTION GUESSER RETURN INTEGER IS
16  2   0 -14   RESPONSE : INTEGER;       -- Player's Hi, Lo, or Equal response
17  2   0   2   COUNT    : INTEGER;       -- Counts the # of guesses
18  2   0   4   GUESS    : INTEGER;       -- the guess
19  2   0   6   LASTGUESS: INTEGER;       -- the previous guess
20  2   0   8 BEGIN
21  2   0   8   COUNT := 1;               -- Initial some stuff
22  2   4   8   GUESS := 0;
23  2   8   8   LOOP
24  2   8   8     LASTGUESS := GUESS;
25  2  12   8     GUESS := (LOWBND + HIBND) / 2;  -- Uses a binary search
26  2  22   8
27  2  22   8     IF LASTGUESS = GUESS  THEN
28  2  28   8        PUTLINE ("You messed up your entries!  I guarantee");
29  2  72   8        PUTLINE ("to guess your number in less than 11 moves");
30  2 118   8        PUTLINE ("but you entered a wrong response! and that");
31  2 164   8        PUTLINE ("really upsets me.  I never make mistakes");
32  2 208   8        PUTLINE ("like that . . .");
33  2 227   8        RETURN COUNT;
34  2 229   8     END IF;
35  2 229   8
36  2 229   8     NEWLINE; NEWLINE;
37  2 233   8     PUTSTR("Guess # ");
38  2 245   8     PUTINT(COUNT);
39  2 248   8     PUTSTR("   Computer's Guess = ");
40  2 274   8     PUTINT(GUESS); NEWLINE;
41  2 279   8
42  2 279   8     PUTSTR("Enter:  1=Too High,  2=Too Low,  3=Equal or Quit ? ");
43  2 334   8     GETINT(@RESPONSE);
44  2 338   8
45  2 338   8     CASE RESPONSE IS
46  2 346   8       WHEN 1 => HIBND := GUESS - 1;
47  2 352   8       WHEN 2 => LOWBND := GUESS + 1;
48  2 361   8       WHEN 3 => RETURN COUNT;
49  2 366   8       WHEN OTHERS => PUTLINE ("Invalid input ignored");
50  2 394   8     END CASE;
51  2 406   8
52  2 406   8     COUNT := COUNT + 1;
53  2 412   8
54  2 412   8   END LOOP;
55  2 412   8
56  2 412   8 END; -- OF FUNCTION GUESSER
57  1 416  91
58  1 416  91
59  1 416  91 BEGIN -- PROCEDURE GUESSNUM
60  1   0  91   AGAIN := TRUE;
61  1   4  91   PUTLINE("NUMBER GUESSING GAME"); NEWLINE;
62  1  30  91   PUTLINE("Choose a number between 1 and 1000 and the");
63  1  76  91   PUTLINE("computer will try to guess your choice.");
64  1 119  91
65  1 119  91   WHILE AGAIN
66  1 119  91   LOOP
67  1 124  91
68  1 124  91     NEWLINE;
69  1 126  91     PUTSTR("Press RETURN after you have chosen your number:  ");
70  1 179  91     GETSTR(@S);
71  1 183  91
72  1 183  91     LOWBND := 1;
73  1 187  91     HIBND  := 1000;
74  1 193  91
75  1 193  91     NGUESSES := GUESSER;    -- Call the guessing routine
76  1 198  91     NEWLINE;
77  1 200  91     PUTSTR("Number of guesses = ");
78  1 224  91     PUTINT(NGUESSES);       NEWLINE;  NEWLINE;
79  1 232  91
80  1 232  91
81  1 232  91     PUTSTR("Do you wish to play again (CR=Yes/N=No) ? ");
82  1 278  91     GETCHAR(@CH);
83  1 282  91     NEWLINE;
84  1 284  91     IF CH='N' THEN AGAIN := FALSE;
85  1 296  91     END IF;
86  1 296  91   END LOOP;
87  1 296  91 END; -- OF PROGRAM
```

End Listing

character. An example definition looks like,

CH : CHARACTER;

A character constant is delineated by single quote marks: 'A' or 'B'. The variable CH is assigned the value of 'A' by entering

CH := 'A';

STRINGs are similar to the strings defined in UCSD Pascal but are not the same as arrays of CHARACTER. The declaration

S : STRING;

gives S a length of 128 characters. Character string constants are enclosed in *double quotes* (note the difference between strings and characters). For example, an assignment might be made to S by entering,

S := "The moon will be in total
eclipse on: ";

However, unlike Pascal, a string in Augusta is not an array of characters. The notation S(*i*) refers to the *i*th element of an array S, not to character *i* within a string. Instead, several "built-in" procedures and functions are provided by Augusta to manipulate strings. These procedures are summarized in Table I.

Table I.

Summary of Predefined Augusta Procedures and Functions.

```
PROCEDURE PUTINT ( I : INTEGER );
     Outputs I to the display.  I may be either a variable or an
     expression.

PROCEDURE PUTSTR ( S : OUT STRING );
     Outputs S to the display.  S may be either a string
     variable, prefaced by the special "@" symbol, or a string
     constant.

PROCEDURE PUTCHAR ( C : CHARACTER );
     Outputs the single character C.

PROCEDURE GETINT ( I : OUT INTEGER );
     Reads an integer typed at the keyboard, returning the
     entered number in I.

PROCEDURE GETSTR ( S : OUT STRING );
     Reads a string from the keyboard, returning the
     input in S.

PROCEDURE GETCHAR ( C : OUT CHARACTER );
     Reads a single character from the keyboard, returning the
     input in S.

PROCEDURE NEWLINE;
     Prints a CR/LF sequence.

PROCEDURE PUTLINE ( S : OUT STRING );
     Equivalent to PUTSTR(@S) followed by NEWLINE;

FUNCTION PEEK ( I : INTEGER ) RETURN INTEGER
     Equivalent to Microsoft BASIC-80's PEEK function, this
     function returns the value of the byte at memory address I.

PROCEDURE POKE ( A, I : INTEGER );
     As above, POKE is similar to BASIC-80's POKE, and is used
     to write the byte value I to memory address A.

PROCEDURE SUBSTR ( S1, S2 : OUT STRING; START, LEN : INTEGER );
     Copies  a  substring from S2,  beginning at byte  START  and
     continuing for length bytes, into S1.

FUNCTION CHAR ( S1 : OUT STRING; POSITION : INTEGER ) RETURN CHARACTER
     Returns the character at byte POSITION in S1.

PROCEDURE APPEND ( S1, S2 : OUT STRING );
     Appends string S2 on to the end of S1.

PROCEDURE ASSIGN ( S1 : OUT STRING; POSITION, VALUE : INTEGER );
     Changes the byte at POSITION in S1 to VALUE.
```

The length of a string is obtained by a reference to the variable's LENGTH *attribute*, as shown below,

I := S'LENGTH;

returns the current length of string S. Augusta also supports an attribute for array variables,

X'LAST

which returns the dimensioned size of array X.

Constant identifiers are declared like variables as,

TEN : CONSTANT := 10;
– symbol "TEN" has a value 10

A constant identifier can be used anywhere that an explicitly written constant is used. For example,

I := TEN + TEN;

would assign the value 20 to I. Note that Augusta does not provide constant strings; only integers, characters, and booleans are permitted as constant identifiers.

Constants may be entered in any base, 2 through 16, by using Ada's base conversion notation. For example,

2#101011

is a constant in base 2.

16#FF

is a hexadecimal constant.

Augusta, like Ada, requires most variables to be defined before they can be used in a program. But here are some exceptions. In particular, a FOR loop control variable, such as I, in

FOR I IN 1..10

is automatically defined as an integer that is *local* to the scope of the FOR loop. Once the FOR loop is completed, the definition of I goes away. At first glance this may seem peculiar, but it answers the classic question: "What is the value of I at the end of the loop?" Is it 10 or 11? By removing the definition of the loop control variable at loop exit, there can be no *ambiguity; I is truly* undefined.

Variables may also be declared local to a BEGIN-END block by using the DECLARE statement as shown in Listing 2. Once the program exits the block, both the definition of the variables and the storage space that they occupy are made free.

In the declarations area of a procedure or in the declarations following a DECLARE statement, the different

types, including constants, may appear in any order. However, no more than 15 identifiers may appear in a list of identifiers as in,

I,J,K,L,M : INTEGER;

The total number of identifiers allowed in a program is dependent on the amount of memory your computer has. Each symbol table entry requires 17 bytes. On the Osborne 1, over 400 identifiers may be declared and simultaneously in use. Identifiers that are "local" to a procedure or function are removed from the symbol table after the procedure or function has been compiled. This means that the total number of identifiers used in a program may be far in excess of 400, provided that most are defined locally to procedures and functions.

Arrays

Arrays are always one dimensional and begin at 0. Example,

X : ARRAY (10) OF STRING;

defines 11 string elements X(0), X(1), . . ., X(10), each up to 80 characters long. Augusta also supports arrays of integers, characters and booleans.

Type Conversions

Augusta, like full Ada, permits variables and expressions to be converted to other types. For example, if you need to obtain the ASCII value of a character variable CH, you type,

I := INTEGER (CH);

The "type" identifier INTEGER is used like a function call. The single parameter CH is converted to an integer. Similarly, to convert an integer 65 into the corresponding ASCII character 'A', you would type,

CH := CHARACTER (65);

Statements in Augusta

The basic assignment statement has already been shown:

I := 35+78;

Augusta supports the usual arithmetic operators, including addition, subtraction, unary minus, multiplication, division and modulus. The complete set of operators is shown in Table II.

The operations follow the usual arithmetic heirarchy as shown below:

(1) *, /, and MOD are applied first.
(2) + and – are applied second.
(3) Relational operators, such as "=" and "<=", appear third.
(4) Boolean operations AND and OR are last.

For example, the expression 3 + 5 * 8 is evaluated as 3 + (5 * 8) because multiplication comes before addition.

Conditional Statements

IF-THEN-ELSE and IF-THEN-ELSEIF logic provide for conditional execution of program statements. As shown by Table III, multiple statements may follow THEN or ELSE without the use of a BEGIN-END block. An END IF or additional ELSE or ELSEIF clause terminates the sequence of statements.

Augusta provides an unusual "short circuit" conditional. In most program-

Table II.

Summary of Augusta arithmetic operations.

```
+     Addition,  as in 3 + 5 gives 8

-     Subtraction, as in 100 - 25 gives 75
      and unary minus, as in -9.

*     Multiplication, as in 10 * 5 gives 50

/     Division, as in 50 / 5 gives 10

MOD   Modulus, as in 7 MOD 3 gives a remainder of 1
      (the remainder of X/Y)

NOT   Unary boolean "NOT".

AND/AND THEN
      Boolean AND

OR/OR ELSE
      Boolean OR

=     equality comparisons

/=    not equal

<     less than

<=    less than or equal

>     greater than

>=    greater than or equal
```

Table III.

Shown here are several examples of Ada's IF-THEN-ELSE/ELSEIF-END IF conditional statements. If IF, except for ELSEIFs, must have a matching END IF.

```
(a)
     IF A=B THEN FOUND := TRUE;
     END IF;

(b)
     IF A=B THEN FOUND := TRUE;
     ELSE FOUND := FALSE;
     END IF;

(c)
     IF A=B THEN FOUND := TRUE;
     ELSEIF A=B+1 THEN EXIT SEARCH;
     ELSEIF A=0 THEN I:=I+1;
     ELSE PUTLINE ("MISMATCH");
     END IF;

(d)
     IF A=B THEN FOUND := TRUE;
       ELSE
       IF A=B+1 THEN EXIT SEARCH;
         ELSE
         IF A=0 THEN I := I + 1;
         END IF;
       END IF;
     END IF;
```

ming languages, the statement,

```
IF (N<>0) AND (J/N=1) THEN...
```

causes *both* conditions to be tested regardless of the outcome of the first comparison. Of course, if N=0 the program will likely halt with a divide-by-zero error. The programmer probably meant to write:

```
IF (N<>0) THEN
   IF (J/N=1) THEN . . .
```

In this form the second condition will only be evaluated when N is not equal to 0.

Short-circuit notation provides a shorthand method of writing this type of conditional. An example is given below (Note: In Ada "/=" is the "not equal" relational operator):

```
IF (N/=0) AND THEN (J/N=1)
   THEN...
```

If N is equal to 0 then the second condition is never evaluated.

Similarly, if any one of several conditions is true, then it may not be necessary to evaluate all the conditions. For example,

```
IF (A=1) OR (B=2) OR (C=3)
   THEN...
```

might be written in Ada as

```
IF (A=1) OR ELSE (B=2) OR ELSE
   (C=3) THEN...
```

If A is equal to 1, none of the other conditions will be tested and control will immediately transfer to the THEN part of the statement.

The short-circuit conditional behaves exactly like a boolean expression. The statement

```
B := (A=1) OR ELSE (B=2);
```

functions similarly to a boolean OR of (A=1) OR (B=2).

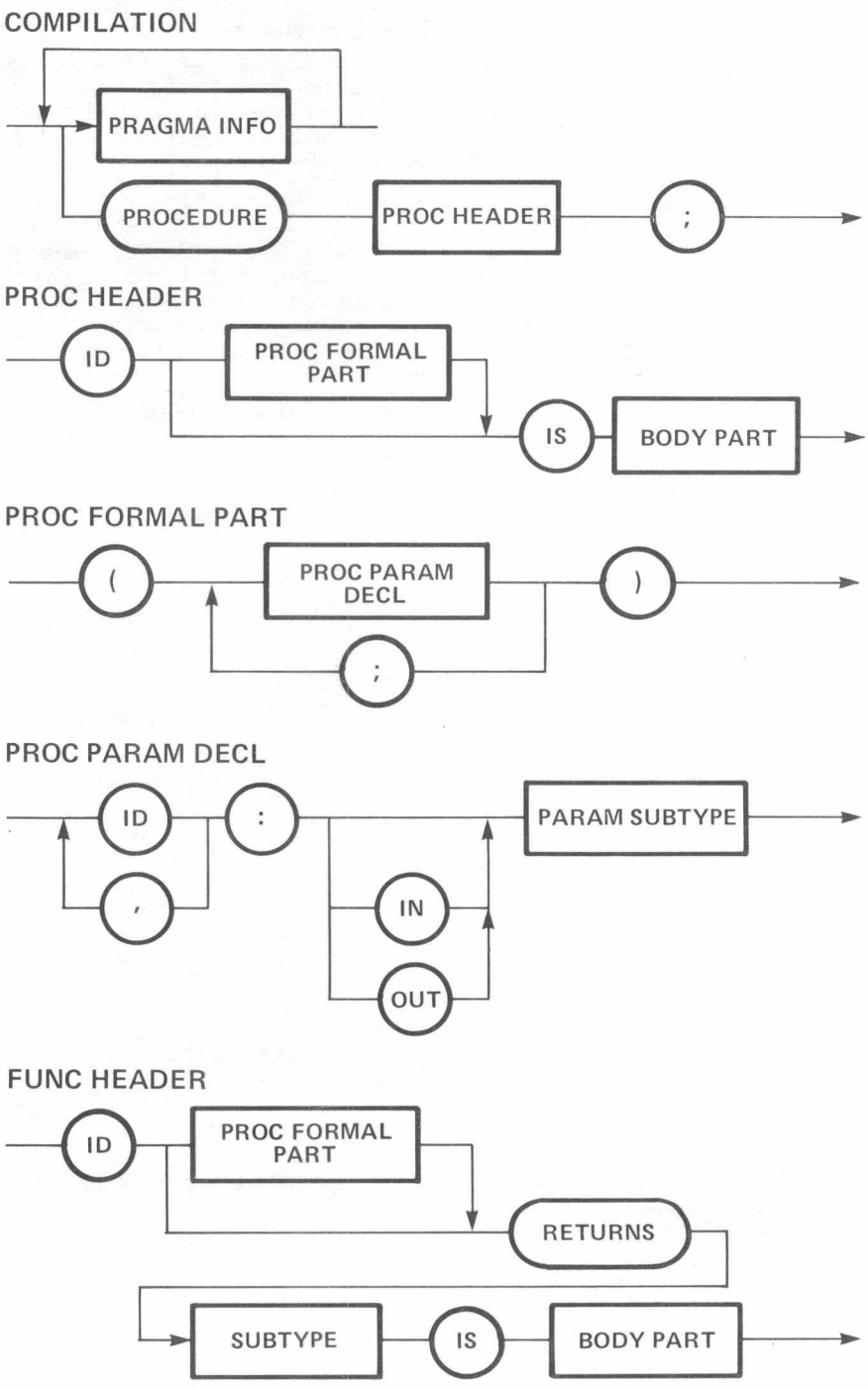
Looping Statements

The LOOP-END LOOP construction forms the basic looping mechanism. The statement

```
LOOP
   – process statements
END LOOP;
```

is an infinite loop. By placing a modifier before the loop, the number of itera-

FIGURE 1



(Figure 1 continued on next page)

tions is restricted. Example:

```
FOR I IN 1..10
LOOP
   – process statements
END LOOP;
```

is a traditional FOR loop, executing the enclosed statements ten times. A FOR loop can also decrement the control variable by specifying the REVERSE option, as in,

```
FOR I IN REVERSE 10..1
```

Note that I is automatically defined as an integer variable that is local to the loop.

A WHILE modifier executes the enclosed loop as long as the control expression remains true. A simple WHILE loop might look like

```
I := 1;
WHILE I<10
LOOP
   I := I + 1;
END LOOP;
```

The loop is repeated for values of I from 1 to 9. On exit, I equals 10. The looping condition is always tested before the loop is executed. A Pascal-like REPEAT-UNTIL loop, where the condition is tested at the end of the loop, is possible and is shown in the description of the EXIT statement.

The CASE Statement

The CASE statement uses the value of an expression to select one of several conditions. For the most part, it is similar to the CASE statement in Pascal. Example:

```
CASE CH IS
   WHEN 'A' ! 'B' =>
                  DOCOMMAND;
      PUTLINE("COMMAND
         A OR B COMPLETED");
   WHEN 'C' => DOCOMMANDC;
   WHEN OTHERS =>
                  DOOTHERS;
END CASE;
```

In the example above, if CH has either the value 'A' or 'B' ("!" separates items in a CASE selector list), then the procedure DOCOMMAND is executed. If CH is neither 'A', 'B' nor 'C', then the OTHERS part of the CASE statement is executed. The OTHERS condition is optional and may be omitted. If OTHERS is not present and the value of CH is not one of the specified choices, then none of the statements is executed.

In the example above, if CH is 'A' then two statements are executed. Multiple statements do not need to be enclosed within BEGIN-END statements as is done in Pascal.

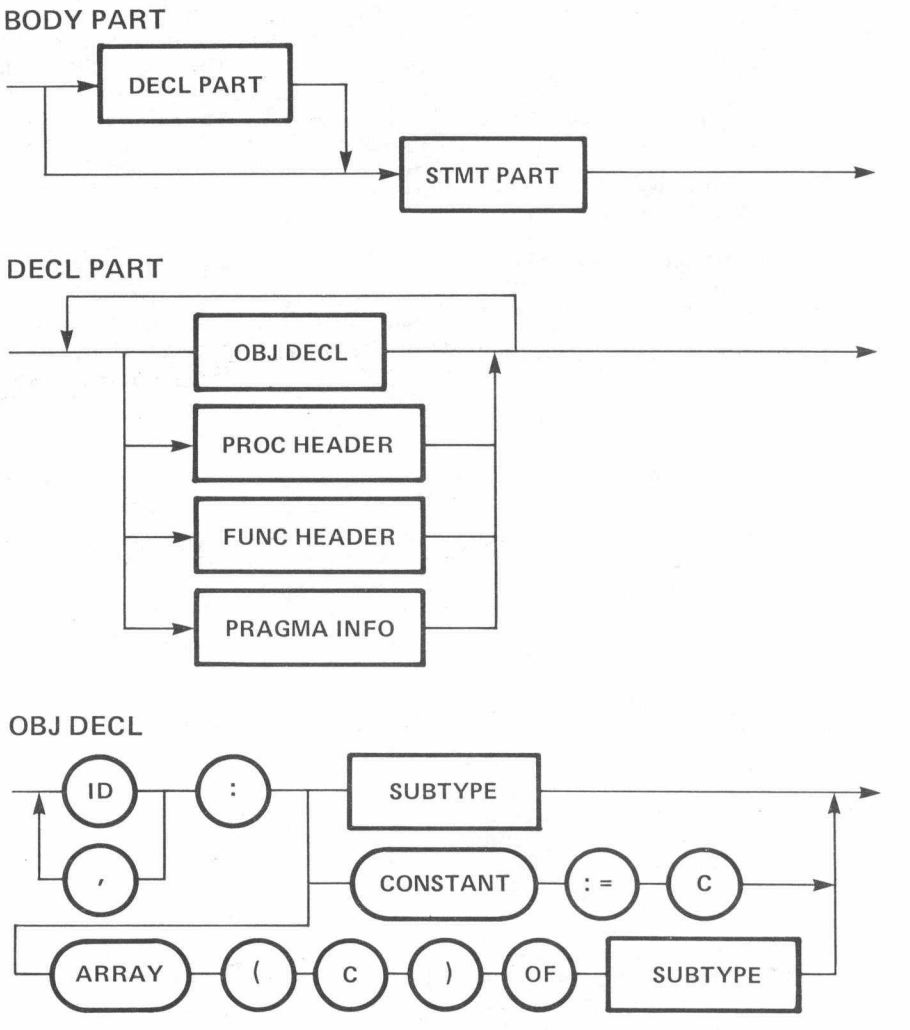
The EXIT Statement

Ada provides an EXIT statement to force an immediate jump out of one or more surrounding LOOPs. Augusta limits the EXIT to only transfer control out of the currently executing loop. For example,

```
FOR I IN 1..10
LOOP
   PUT_INT(I); NEW_LINE;
   EXIT WHEN I=5;
END LOOP;
PUT_LINE ("Out of Loop");
```

When I has the value of 5, control immediately transfers to the PUT_LINE following the END LOOP statement. The conditional WHEN clause is optional and may be omitted. The statement then becomes:

(Figure 1 continued from previous page)



(Figure 1 continued on page 22)

```
FOR I IN 1..10
LOOP
   PUT_INT(I); NEW_LINE;
   EXIT;
END LOOP;
```

which prints 1 and immediately terminates the loop.

The EXIT directive may be used in combination with a LOOP-END LOOP block of statements to perform a Pascal-like REPEAT-UNTIL. In the REPEAT statement, the loop is always executed at least once since the condition test is performed at the end of the loop. In Augusta, a REPEAT-UNTIL type statement is coded as

```
I := 10;
LOOP
   PUT_INT(I); NEW_LINE;
I := I + 1;
EXIT WHEN I>5;
END LOOP;
```

The above will always print the initial value of I.

Pragma Compiler Control Comments

"Pragmas" are a special type of comment statement that acts as an instruction to the compiler. Augusta uses pragmas to "include" additional source texts during the current compilation and to control the printing of the compiler generated source listing.

During compilation, Augusta can temporarily stop reading the primary source file and begin reading another source file. When the end of the second source file is reached, the compiler resumes reading the original file at the point where it was suspended. An "include" pragma has the form

PRAGMA INCLUDE ("Filename");

Included files may, in turn, include other files. The nesting level is determined by the maximum number of files that BASIC allows open at one time.

Procedures and Parameter Passing (Table IV)

The basic program unit is a PROCEDURE or FUNCTION that optionally contains its own set of "local" variables. Memory space for local variables is allocated when the procedure is called and is made free when the procedure returns to its caller.

Parameters may be passed as either "IN" or "OUT" type. IN parameters are passed by value. That means that the called procedure is given a copy of the data to process. An assignment to an IN parameter variable has no effect outside the procedure. Conversely, if an OUT parameter is assigned a new value, it is passed outwards to the calling procedure.

An "@" symbol before an identifier in the parameter list tells the compiler to pass the parameter by reference rather than by value. The "@" symbol must be used when the parameter is defined as OUT. For reasons that will be explained in Part 4 (scheduled for the June, 1983 issue of *DDJ*), the compiler does not perform any type checking of parameters at the point of the procedure call. In addition, string variables and arrays must always be passed by reference – never by value. The following statement illustrates the passing of a variable reference to a procedure by preceding it with the "@" symbol:

```
SEARCHROUTINE ( ELEMENT,
      @FOUND );
```

where FOUND is a boolean variable set to TRUE or FALSE by the search routine.

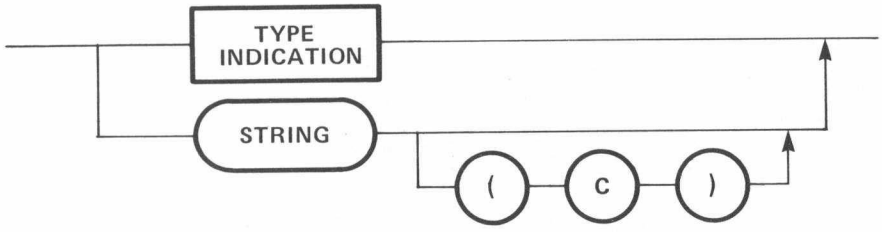
Table IV.

Parameters to procedures may be passed either "by value" or "by reference." When passed "by value," Augusta passes a *copy* of the parameter to the procedure. When "by reference," the *address* of the variable is passed. A parameter declared as "OUT" type is treated as a pass by reference variable. If the parameter variable is assigned a new value, that new value is passed outwards to the caller. The program shown in (a) demonstrates pass by value and pass by reference by producing the output shown in (b).

```
(a)

PROCEDURE DEMO IS
  X : INTEGER;

PROCEDURE BYVALUE ( I : INTEGER ) IS
BEGIN
  PUTSTR("ON ENTRY TO BYVALUE, I = ");
  PUTINT(I); NEWLINE;
  I:=1;   -- THIS CHANGE WILL HAVE NO EFFECT ON THE PARAMETER
          -- PASSED TO PROCEDURE BYVALUE
END;

PROCEDURE BYREFERENCE ( I : OUT INTEGER ) IS
BEGIN
  PUTSTR("ON ENTRY TO BYREFERENCE, I = ");
  PUTINT(I); NEWLINE;
  I:=1;    -- THIS WILL CHANGE THE VALUE OF X IN THE CALLING ROUTINE
END;

BEGIN
  X:=100;

  BYVALUE(X);
  PUTSTR("AFTER CALL TO BYVALUE, X STILL = ");
  PUTINT(X); NEWLINE; NEWLINE;

  BYREFERENCE(@X);
  PUTSTR("AFTER CALL TO BYREFERENCE, X IS CHANGED TO = ");
  PUTINT(X); NEWLINE;
END;


(b)

ON ENTRY TO BYVALUE, I=100
AFTER CALL TO BYVALUE, X STILL = 100

ON ENTRY TO BYREFERENCE, I=100
AFTER CALL TO BYREFERENCE, X IS CHANGED TO 1
```

(Figure 1 continued from page 20)

SUBTYPE



PARAM SUBTYPE

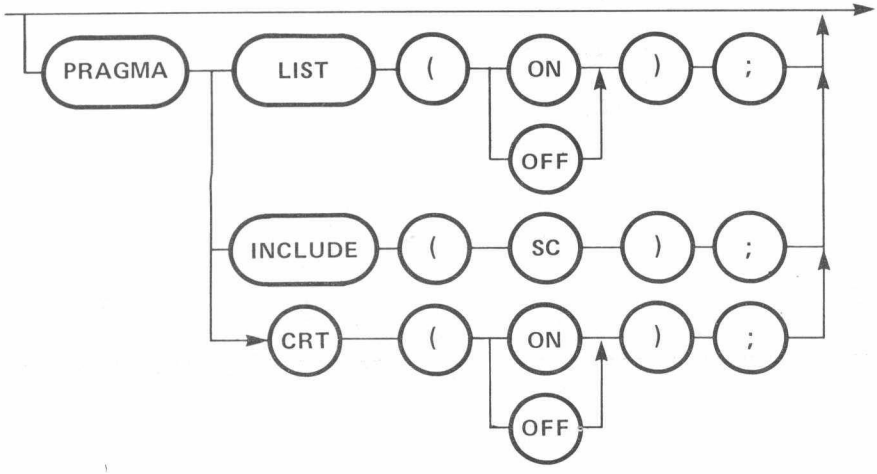
TYPE INDICATION

DEFINED IN SYMBOL TABLE

PARAM TYPE

DEFINED IN SYMBOL TABLE

PRAGMA INFO



(Figure 1 continued on page 24)

An exit from a procedure may be made at any point by using either the EXIT or RETURN statement. To exit a procedure, type

```
RETURN;
```

Functions must always terminate with

```
RETURN <Expression>;
```

where <Expression> is the value of the function to be returned to the caller. For example,

```
RETURN I+10;
```

gives the function the value of I plus 10.

Augusta does not check that each function contains a RETURN statement. It is the responsibility of the programmer to insure that each function definition exits through the RETURN statement. Note also that a function may only return an integer, character or boolean. It may never return a string or an array.

Arrays as Procedure Parameters

Arrays may be passed as OUT type parameters to procedures. If X is defined as

```
X : ARRAY (10) OF INTEGER;
```

then it may be passed to a procedure as

```
PROC2 ( @X(0) );
```

which passes the address of array X. In procedure PROC2, the parameter is defined as

```
PROCEDURE PROC2 ( R : OUT
    INTARRAY );
```

Within PROC2, R is treated as an array of integers.

INTARRAY is a special type declaration, which, along with STRARRAY, CHARRAY and BOOLARRAY, is used to pass arrays as procedure or function parameters. These four types must *only* be used in a procedure's parameter declarations. They are equivalent to

```
ARRAY (?) of <type>
```

where "?" indicates an undefined array size, and <type> is INTEGER, BOOLEAN, CHARACTER, or STRING.

For string arrays, the default string size is 80 bytes. For a *different size* string, an optional string length is specified. For example,

```
PROCEDURE PROC2 ( S : OUT
    STRARRAY (120) );
```

defines S as an array of strings, where each S(i) is up to 120 bytes long.

In the March Issue

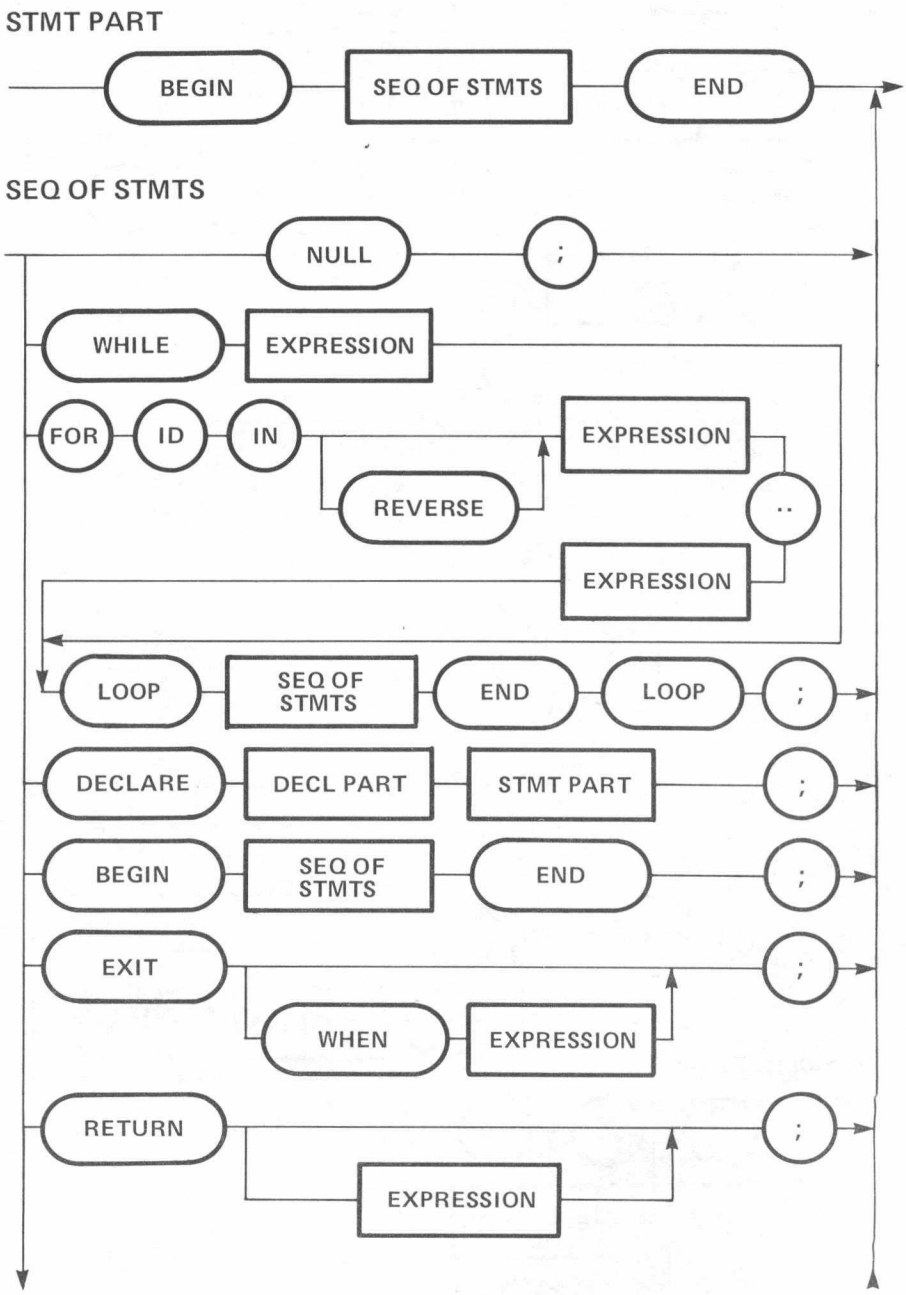
Part 2 describes how "stack machines" operate and how they are used to perform expression evaluation and procedure calls in a language like Augusta. Augusta's p-machine consists of about 80 instructions, ranging in length from a single byte up to 7 bytes. Each of these opcodes will be described in detail.

DDJ

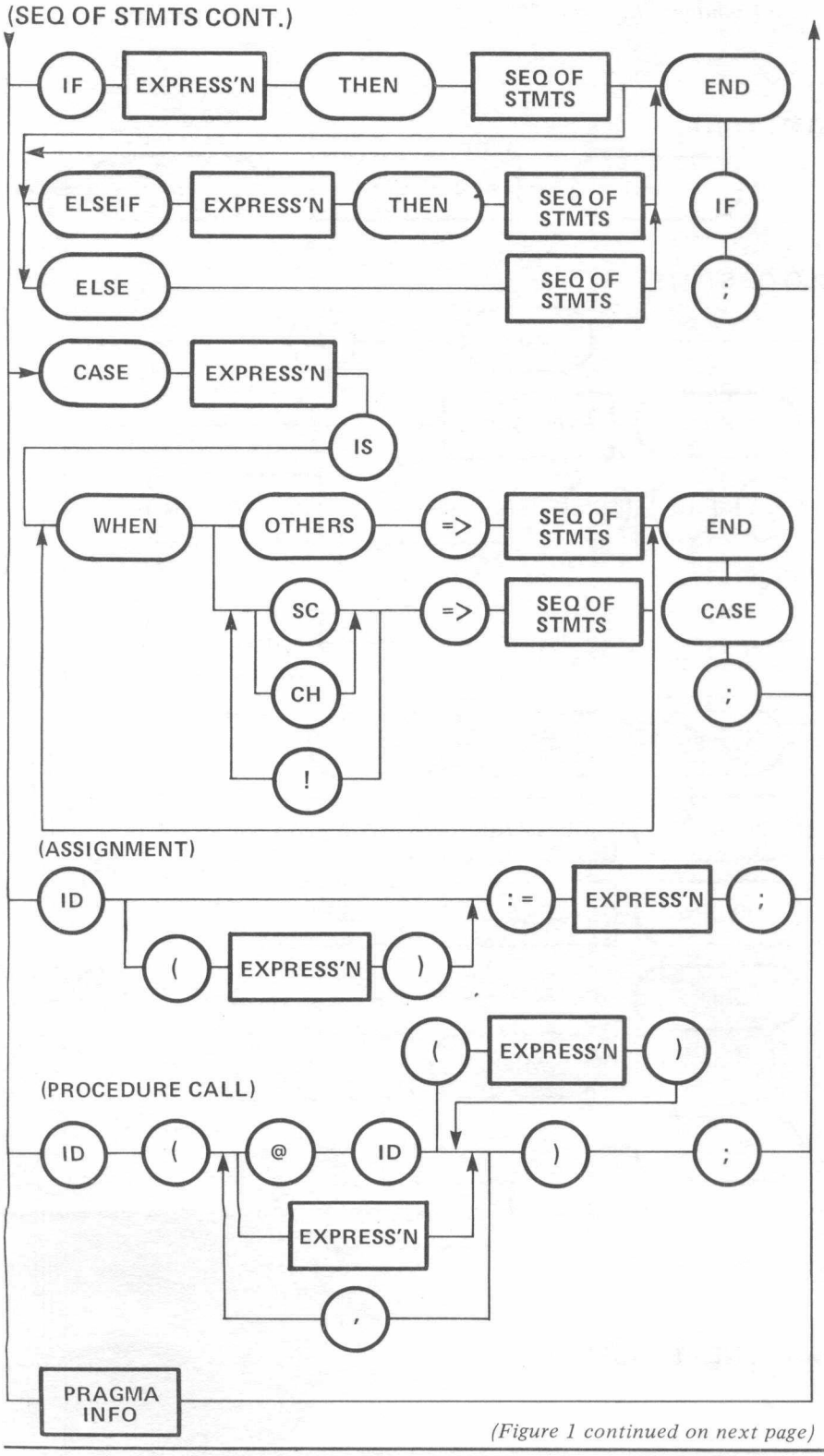
References for Parts 1 through 4

1. *Reference Manual for the Ada Programming Language*, United States Department of Defense, July 1980.
2. Kin-Man Chung and Herbert Yuen, "A Tiny Pascal Compiler, Part 1: The P-Code Interpreter," *BYTE*, Sept. 1978, Vol. 3, No. 9, p. 58.
   – "Part 2: The P-Compiler," *BYTE*, Oct. 1978, Vol. 3, No. 10, p. 34.
   – "Part 3: P-Code to 8080 Conversion," *BYTE*, Nov. 1978, Vol. 3, No. 11, p. 82.
3. *UCSD Pascal Version 1.5 (Sept. 1978)*, Institute for Information Systems, Kenneth Bowles, Dir.
4. Frank De Remer, Tom Pennello, and W. M. McKeeman, "Ada Syntax Chart," *SIGPLAN Notices*, Sept. 1981, Vol. 16, No. 9, p. 48.
5. Alred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design* (Reading, Mass.: Addison-Wesley Publishing, 1977).
6. Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report, 2nd Ed.* (New York: Springer-Verlag, 1974).
7. Terrence W. Pratt, *Programming Languages: Design and Implementation* (Englewood Cliffs, NJ: Prentice-Hall, 1975).
8. Alan C. Shaw, *The Logical Design of Operating Systems* (Englewood Cliffs, NJ: Prentice-Hall, 1974).
9. *MBASIC – Microsoft BASIC-80* (Reference Manual), (Hayward, CA: Osborne Computer Corporation).
10. Bonet, R, *et al.*, "Ada Syntax Diagrams for Top-Down Analysis," *SIGPLAN Notices*, Sept. 1981, Vol. 16, No. 9, p. 31.

(Figure 1 continued from page 22)

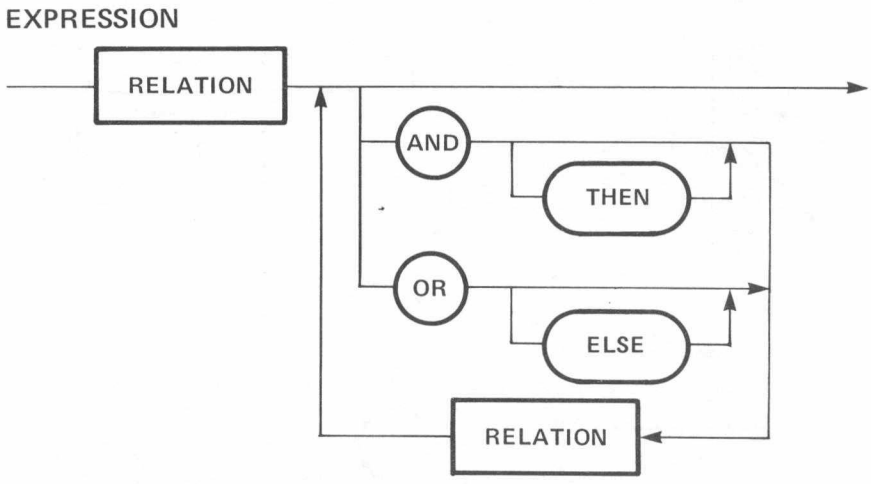


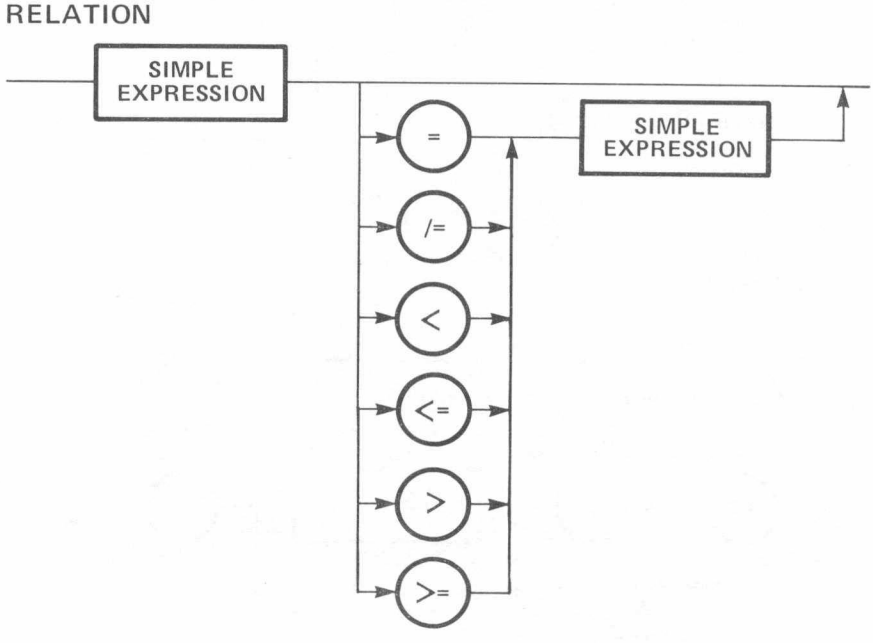
(CONT. NEXT PAGE)



(Figure 1 continued on next page)

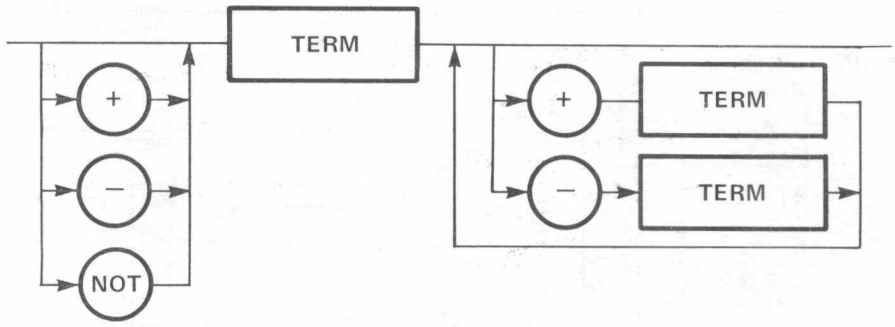
(Figure 1 continued from previous page)



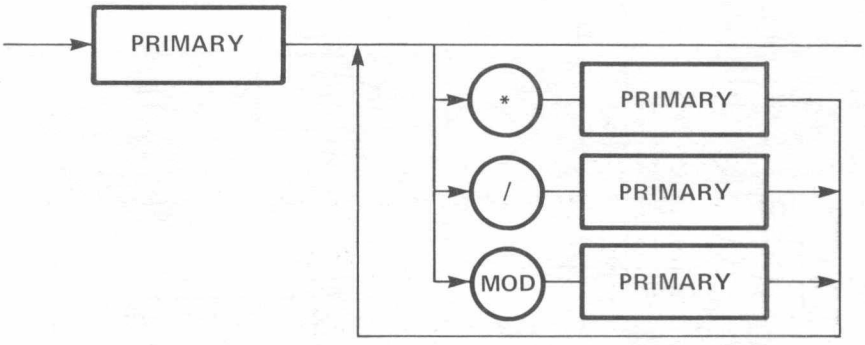


(Continued on next page)

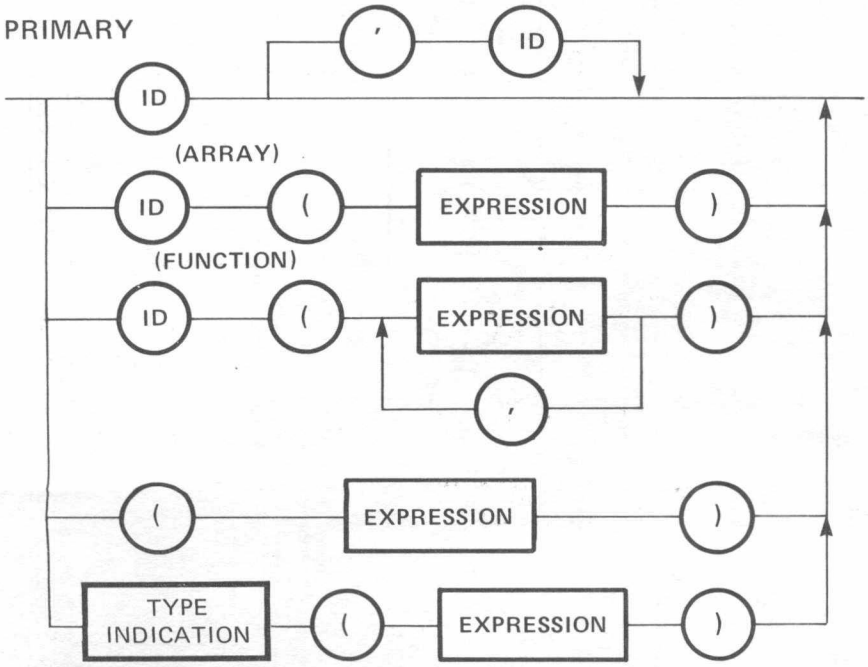
SIMPLE EXPRESSION



TERM



PRIMARY



(End of Figure 1)

XANADU

Hypertext from the Future

by Roger Gregory

Roger Gregory, c/o XOC, Box 7615, Ann Arbor, MI 48017.

The Xanadu Hypertext System is a unified system for storing and managing certain forms of information, namely: bodies of documents. It is built around a small number of relatively simple concepts:

- DOCUMENTS
- Multiple VERSIONS of Documents
- Distributed Storage and Distributed Computation – NETWORKING between main frames and microcomputers
- Isolation of storage functions from display functions–BACK-END/FRONTEND

I will discuss each of these in turn and thereby build up a picture of what the Xanadu System is, what it does, and what makes it unique.

Documents

A "document" is a chunk of contiguous characters (or other data) which functions in some way as a conceptual unit. It is simply a piece of text which some user has declared to be a single thing. It corresponds roughly to the notion of a "file" as found in most computer systems. Beware of this analogy, however, because it breaks down quite thoroughly in the face of the other properties of the system.

A document may contain whatever the user decides to put in it. While documents will generally contain text, the system assumes nothing about their contents. Any arbitrary set of data which can be represented as a string of bytes may be stored in a document. Possible documents include letters, books, chapters of books, memos, marginal notes, musical scores, reports, pictures, computer programs, maps, diagrams, recordings, etc.–whatever the user desires the document to be.

Each document stored in the system is assigned a unique identifier when it is created. This identifier can then be used in later transactions with the system to designate that particular document. This document identifier is a special sort of number called a "tumbler." Tumblers are multi-part numbers similar to those used to number sections of technical manuals and to index books in the Dewey Decimal System. For example, 1.0.3.15 and 123.5.6391.0.8.1 are both tumblers. They have the property that they can be expanded at any point, and it is therefore possible to implement a numbering system that never runs out of precision. This means that there is no limit to the number of document identifiers possible and therefore no theoretical limit to the number of possible documents due to a finite address space.

Since a document is a linear array of bytes, we can refer to any particular character in a document by its position in the document. Thus, by combining a document's unique identifier with a character position within that document, we can uniquely address any character stored in the entire system.

By designating a particular (starting) character and a length we can address any contiguous string of characters in the system. Such a string is called a "span." By clever use of tumblers instead of plain integers to represent character positions and span lengths, it is possible to have spans which contain whole documents and cross document boundaries.

Versions

The Xanadu Hypertext System is a real and currently available product which can manage multiple differing versions of a single document. Typically, documents will change with time, as they are corrected or revised. Parallel versions may be made, as when writers weigh alternate drafts of a chapter or when a computer program and its documentation are maintained for several different installations.

Whenever a document is changed, the changes are recorded. The system remembers both the old and the new versions and can retrieve either on command. Thus it is possible to retrieve a document in the state it was in at any point in time since its creation. This can be particularly useful, for example, in software development or creative writing when one may make false starts and wish to back up to a previous state. In addition, the system can indicate the changes which caused two alternate versions to differ. Thus a complete "audit trail" through the history of a document can be constructed. Such a trail is in the form of an historical trace tree which branches whenever parallel versions are maintained.

In the general case, the problem of comparing two pieces of text to determine the differences between them is NP-complete. It may seem curious then that the Xanadu System is able to do this. The reason that the system can determine how documents differ is because *it does not compare them*. Rather, *it* performs the actual changes and "remembers" them, instead of having a whole new document passed to it by some external editor, as most systems would do.

Xanadu understands a simple set of editing primitives by which it can be directed to make changes to a document. Those are:

- INSERT characters at some point in the document.
- DELETE characters from some point in the document.
- MOVE a span of characters from one location in the document to another.
- COPY a span of characters from one location in the document (or from anywhere in the rest of the system, for that matter) to some location in the document.

A moment's thought will reveal that these operations are sufficient to perform then more complex editing operations.

This sort of arrangement is not unique. However, most systems which do this, such as Bell Laboratories SCCS (Source Code Control System) and DEC's CMS (Code Management System) merely keep a log of the edit operations and then perform those operations in chronological order to produce the desired version. While this is sufficient in many cases, it is unwieldy when there are many parallel branching versions and it is costly in terms of computational overhead, becoming more so as more and more changes are made to a document.

In the Xanadu System, these primitives correspond to operations on the proprietary data structures in which the documents are stored. These data structures tell the system which characters are in which versions and in what order. This allows the proper retrieval of any version quickly and efficiently. The historical traceback facility is built in implicitly. To the best of our knowledge, the Xanadu System is the only system in existence able to manage multiple versions of data in such a general and efficient manner.

Links

A "link" is merely an explicit logical connection between one piece of data and another. The Xanadu Hypertext System supports a completely generalized

linking facility. Any body of text in the entire system can be linked to any other body of text in the system. Links may connect text within a document or run between documents.

Any given link connects from one set of text in the system to another. The bodies of text at either end of the link are therefore called the link's "end-sets." The first end-set is called the "from-set" and the second is called the "to-set." By convention links have an implied direction from the from-set to the to-set, not surprisingly, although the relationship is entirely symmetrical and a link may be followed in either direction.

The potential contents of an end-set are totally unrestricted. An end-set may consist of an unlimited number of spans. The characters linked to or from need not be contiguous or even in the same document.

Links themselves are said to reside inside documents, located in a separate logical address space from the text. A link may reside in one document and link from a second document to yet a third. The location of the residence of a link is entirely independent of the contents of the link's end-sets. The fact that links are among the potential contents of documents means that links themselves may be linked from or to, just as characters may. Thus very general sorts of indirect structures may be assembled.

It was mentioned above that the end-sets are symmetric and only distinguished by convention. In fact, a link may have any number of end-sets. The use of these extra end-sets may vary depending on the purpose of the link. We have found it convenient to define, again by convention, a third end-set called the "three-set."

The purpose of the three-set is to designate information pertaining to the link itself. By this means we can define "link types" which may in turn be used to determine how the link is to be interpreted. For example, we might create a "quotation link" type, which designates that the passage linked to is to be displayed in quotation marks at the location linked from. A "footnote link," on the other hand, might simply designate that a superscripted asterisk be shown at the "from" location and that the to-set contents be displayed only when the user so requests. In a more advanced system, the three-set might point to a program or subroutine to be downloaded into the display computer. This program would then decide how to display the link and its end-sets. In any case, the mechanism is totally general and constrained only by convention.

There are very few systems which attempt to implement a link facility at all like this. We are aware of only one system which even comes close, Englebart's "Augment," available on Tymenet. Augment implements links by embedding code sequences directly into the text. This technique has a number of drawbacks: it rapidly becomes unworkable when more than a small number of links are defined, it is very difficult to maintain the bi-directionality of links using such a scheme, discontinuous end-sets are just about impossible, and embedded code sequences would interfere with the versioning mechanism. As with versions, proprietary data structures enable our system to work. Xanadu links are not embedded in the text. Although links are said to reside in documents, this is simply a means to allow links to be addressed and thus linked to. There are a number of tricky problems that a linkage system such as this presents. In particular, answering the question, "What links connect from other places to a particular location?", is very difficult using most conventional schemes of organizing data. Consider, for example, the problems inherent in assembling such things as citation indices. Nevertheless, we believe we have the general-case solution to these problems.

One might ask what happens to the end-sets of links when the documents containing the characters in the end-sets are changed. There is a very close relationship between the linking facility and the versioning facility. When new versions of a document are created, links are preserved. It is possible to link to a particular version of a document, to all versions, or to the most current version. Links will always point to the same text even though the document containing the text may change. For example:

Here is some text.	Some text in a document.
Here is some more text.	Some characters are inserted.
Here is me more text.	Some characters are deleted.

[The underlined characters represent an end-set of some link.]

The ability to maintain links across multiple versions of documents is one of the totally unique and most powerful aspects of the Xanadu System. The problems inherent in doing this are almost unimaginably subtle, but we believe we have found the solution.

Multi-user

The facilities of the Xanadu Hypertext System would be of limited utility if each user functioned in isolation. Xanadu is therefore designed to be a multi-user system, allowing independent users to access and manipulate the same body of information.

Each user in the system has a unique user identifier tumbler. Each document in the system has an owner, and the owner's user identifier tumbler comprises one of the fields of the document identifier tumbler. By this means we can immediately determine the ownership of any document.

The owner of a document may grant other users permission to access it. As with most aspects of the Xanadu system, the access privilege mechanism is as general as possible. Permission to access a particular document may be granted to an arbitrary number of individuals, to groups of users,or to the entire user community. Type of access permitted (Read, Write, etc.) is not a relevant concept in the Xanadu System. You may always edit another user's document to which you have access, but in doing so you create a new version which is your own document. By this means we can immediately determine the ownership of any document.

The multi-user capability of Xanadu makes a number of useful activities possible. For example, you can write marginal notes commenting on someone else's work. You write your comments in a document of your own and then link them to the other person's document. The other user need never be aware of this. It is like scribbling in a library book without defacing the book to the eyes of others. As another example, it is possible, using documents, links, and the permission facility, to create a quite sophisticated "electronic mail" system. By passing a link to another user you can point him at something you want him to read.

Networking

As the number of users on an installation of the Xanadu System grows larger, the processing and storage capability of even the largest and fastest computer will eventually be used up. The only viable solution to the fundamental resource limitations of centralized facilities is distributed processing and storage. The Xanadu System is designed to be able to grow arbitrarily large without having a significant adverse impact on overall system performance. It does this by allowing the system to spread out over as many machines as are needed to support all users effectively.

The portion of the system which stores and retrieves documents and links is designed to be able to run simultaneously on an arbitrary number of machines which are tied together in a network by some sort of communication medium. If a piece of information is requested at one node of the network but is not stored at that node, the Xanadu software running on that node's computer sends a request via the network to the machine in which the information is stored. This machine in turn sends back the requested data.

The system does not care what communications technology is used, as long as it gets the bits from one place to the other. No particular network topology is assumed, as long as paths exist between any nodes which might wish to communicate. Nodes may be added to or deleted from the network continuously with the minimum possible disruption. If a node is deleted, data that was stored solely on that node becomes unavailable, of course, and any communication paths that went through that node may need to be rerouted, possibly delaying some transmissions.

Communications routing from one node to another is automatic and implicit in the network design. Nodes are not required to have full knowledge of the entire network, thus the network can grow without every node being informed of every change to the network topology.

Various optimizations are built in. If a node finds that it is being given frequent requests for data stored on a distant node, it can make a copy of that data and store it locally. The system does not mind that there may be multiple copies of some information. If a request for such information is made, the system tries to get it from the closest or most quickly accessible source. If changes are made to one copy, the system keeps track All of this is accomplished internally by the same data structures that manage all the other functions of the system.

Backend/Frontend

All of the capabilities described above are part of what is called the 'backend." The backend handles storage, retrieval, versioning, linking and editing of data. However, it "knows" nothing about the data, *per se*. The backend is the portion which does all the "impossible" things. It contains the proprietary components which we have developed.

User interaction—the display of old data and the acceptance of new data—is accomplished by the other major portion of the system, the "frontend." The frontend worries about whether the data is text or pictures or music or whatever. The frontend decides how the data is to be formatted for display. The frontend knows how to interface with various input or output devices.

The frontend and the backend communicate using a standardized request and data transmission protocol. The frontend sends commands and data to the backend which responds with other data and status information. The specifications for this protocol are free for the asking to anyone. The frontend will, in general, be on a separate computer which is connected to the backend by a communications line (not to be confused with the communications lines which connect the Xanadu network—the whole network *is* the backend). Any program which understands the protocol, running on any computer connected to the backend, can be a frontend. The frontend need not be designed by us, written by us, or purchased from us. While we will be producing and selling frontends, we expect others to do so as well.

Frontends may be general purpose or special purpose. Different applications, different forms of data, different display and input devices and different user preferences can all be accommodated. For example, one might create an office automation frontend which handles text processing, electronic mail and project management functions through Xanadu, and does accounting and database stuff locally. A software development frontend might use Xanadu to manage source and object code and documentation and have local capabilities for compiling, executing and debugging programs. As a third example, a CAD/CAM frontend might use Xanadu to store drawings and documentation and have simulation and modeling functions built in. All of these different activities can be accommodated using the same backend design—the backend is application independent.

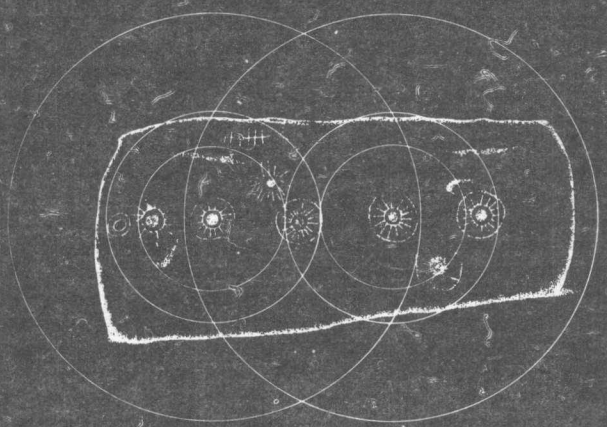
Summary

The Xanadu Hypertext System is one of the most powerful systems in existence for managing text in a micro-to-mainframe database environment. It can store arbitrary numbers of documents and retrieve them on demand. It can organize them by using a highly generalized link facility and by allowing (and keeping track of) multiple users on the same system, and allow those users to work with each other on a common document base. It can grow indefinitely over a large distributed network with minimal degradation in performance. It can accommodate almost any conceivable display or input device or mode of user interaction by separating these functions into an independent frontend.

DDJ

Stone Age Computers

6,000 Years of Computing Science



by Hank Harrison

Human beings are natural computer users. From the first presence of any substance or essence which may loosely be called "mind," human beings used computers.

For centuries scientists thought that humans were the only tool-using life forms on earth, but the "New Archaeology," functioning within the ethical framework set up by the anthropologist Franz Boas, and the highly respected field of Ethology, with its intensive observational techniques, have shown that chimps use straws to pick ants out of tree stumps, and certain bird species use tools for nest building. Nor are human beings the only computer-using species on earth. It is difficult to imagine geese migrating by stellar navigation without some reference to a deeply embedded genetic instinct computer which helps them judge wind speed, direction and endurance.

The annual activities of swallows and butterflies, seals, whales, grunion, even the lowly potato, all seem to be controlled by some sort of internal biocomputer. Von Uxeuell and Tinbergen were awarded a Nobel Prize for proving that bees possess incredible navigational skills and the ability to translate experience through agitated dance activity.

Prehistoric peoples were zoologists. Undoubtedly, Ice Age human beings made close and systematic observation of animals, as each species made use of its well-developed instinct computer. Eventually humans realized that they too had computational abilities. This insight, more than any other, seems to have acted as a stepping stone from which other human beings rose to dominate the planet, and from that remote point in history on, for better or for worse, the human being and the computer were married.

The computing tool was probably one of the first tools, and the human use of external computational aids is probably more than two million years old. In the beginning, stars and planets, moon and sun, wind, rain and thunder dominated the human mind; but as our species spread through the various evolutionary stages, from Homo Robustus the survivor to Homo Sapiens the wanderer and finally to Homo Faber the manufacturer, we developed an ever-increasing ability to focus our own minds to solve problems.

With the advent of computing skills we learned to use wood, wool, crystal, stone, leather, rope, reeds, nut hulls, fruit rinds, sticks, berry juice, sea shells and bones from every conceivable animal, including the human animal, to create knives, hammers and computational aids with which to observe the sky and its complex activities. These computer tools were used, in turn, to calculate the fertility cycles of wild and domestic herds, to tame the environment and to time plant rotation cycles.

In prehistoric times each subgroup, tribe, or family unit obtained knowledge of other similar units in the course of normal migratory behavior. The structure of the Paleolithic social assemblage was supported not only by the shared experience of natural sign and symbol, but also by possession of an incredibly long folk memory stretching back into primordial timelessness, a not so "tabula rasa" genetic brain full of molecular legends and chemophysical icon.

Most human iconography can be traced to a non-racial, species-specific set of images which relates to basal survival realities in the history of the species. Psychotropic substances have taught us about expanded *consciousness* on a personal and subjective level but the "New Archaeology" is demonstrating this molecular mind hypothesis in quantifiable terms. It seems archaic human beings were "Transcendental Computists."

One possible high point in the history of the species is referred to as the Magdalenian epoch – the age when tools

Illustration to left: The Stone of the Seven Suns – Dowth, Ireland.

Illustrations: Martin Brennen

became art objects, the age when the computer was put to extensive use, the age of the discovery of the sun spiral. One might assume that this grand era took place a few hundred years ago, but guess again. The Magdalenian Age centers on 10,000 B.C.! In other words, 12,000 years ago. In Western Europe, and elsewhere on this tiny planet on the edge of the ice sheets, numerous tribes of wandering hunters and gatherers were connected to each other by common symbols, smoke signals, navigational skills, arts, crafts and legends.

Each culture undoubtedly made occasional contact with drifting sailors lost at sea, exiles, exploratory expeditions, and merchants of sorts. Each contact resulted in a transmission of technology, linguistic and symbolic. All of the clans labored under the vagaries of the same weather conditions, the same tides, the same sun and moon, so there is a likelihood that certain basal expressions rose spontaneously at random. These multiple tribal clusters, enriched by primordial stories and legendary skills, seem to have been regulated, especially in the Middle Stone Age, by computational instruments of various kinds, all geared to the sky.

Approximately two million years ago, solar, stellar and lunar observations were made by sand and stick dials. Odd radial patterns were carved in the sand and a tall stick was erected in the center to act as a shadow caster or gnomon. The bush people of the Kalihari, whose ancestry disappears into history and is so old as to be untraceable, still use a simple stick to judge the proper time of day to track quarry. The shadow stick is magic and is shrouded in secret initiation. We now refer to these sticks as sundials, but their full meaning and origin is imperfectly understood.

The period of history vaguely spanning the five thousand years between 10,000 and 5,000 B.C. is called the Mesolithic or Middle Stone Age. Here the Paleolithic "Old Stone Age" art of computing for the hunt was raised to a religious science which was carried over to the more settled Neolithic or "New Stone Age." The art of the Neolithic might be thought of as architectural, except that the Neolithic Architect retained certain elements of Mesolithic shamanistic practice which might today be thought of as psychological. Here at the Dawn of Science, magic was replaced by reason, human beings began to quantify ancient myths, molding them into permanent temple monuments, for ritual purposes.

In approximately 5000 B.C. the ancestors of the fishing and hunting tribes, who had settled into communities, began writing their newly discovered computer formulae in a natural stone language for the edification of their children, and to leave messages for future generations to decipher. These puzzles and messages are written in a code. These not-yet-fully-deciphered codes, when seen in context, seem identical to computer operands and are obviously connected to celestial mechanics in some way.

The verdant hills and valleys of Brittany, Portugal, Wales, Ireland and Scotland, situated close to seas overflowing with fish, were ideal ecosystems for the development of stable societies, computers, and the requisite increase in sophistication in celestial observation and timing techniques that made settlement possible. The computer languages, the symbols and the alphanumeric codes of the New Stone Age derived from

Entrance Stone – New Grange

The human use of external computational aids is probably two million years old!

the hand sign language of the Cro-Magnon cave painters, became the language of the trees of the hunter-gatherers and finally the stone, copper and bronze language variations of gods such as Ogmios, the god of speech, and Lugh or Ra, the god of light. The ancient language of shadow, sun and star light was based on geometry in the cycles of nature and is celebrated in the stones called Megaliths, which support and surround the vast mounds and Star Temples dotted across the maps of France, Germany, Holland, England, Ireland, Scotland and North America.

The Stone Temples and permanent computer monuments had a stabilizing influence on certain societies. These became rich and charismatic even before metal was introduced to the Atlantic Rim cultures. The legends of the new "agra-civitates" which began to flower on the horizon of the 4th millenium B.C., a civilization which Plato called "Atlantis," replaced the legends of the shadow caves, but the basal iconographic systems, the mnenonics of the ancient computations, were passed on.

The eidetic memories of the Ice Age migrations and the Neolithic Temple Computers were dimmed by the hierarchic growth of urban life, but the folk memories of ritual and seasonal behavior akin to the behavior of certain animals remained indelible – indelible because the language of survival and seasonal prediction, the language that made wealth and urban well-being possible, the language that even allowed conceit, was inexorably linked to the secrets of the computer temples. No matter how dim or how repressed, the language of the old gods survived.

A critical point in the evolution of the human use of computers arises in every culture, and has done so since the Ice Ages or earlier. Hundreds of computational experiments from hundreds of locales stand as evidence. The Pyramids, Machu Picchu, the Yucatan, Stonehenge, the Ziggurats of Babylon, the Gothic Cathedrals and the Masonic ground plan of Paris or Washington, DC, are all products of computational preplanning at different moments in history, computational abilities which took celestial mechanics into account.

Anyone with enough air fare and gumption can visit three of these ancient and most ingenious computers on their next visit to Ireland. These three giant stone mounds, located along the River Boyne, four miles inland from the Irish Sea, are undoubtedly among the most magnificent computers in the world. They are older than the Pyramids of Egypt, they are more beautifully situated than the Oracle at Delphi and are more dynamic, more relational, and richer *in culture than* Stonehenge.

The three mounds of Brug na Boinne are located less than one mile from each other and were built in different stages at different times, from at least 4200 B.C. to 3200 B.C. – Knowth and Dowth being the oldest, New Grange the youngest – all according to a preconceived plan, a mysterious building plan that took one thousand years to execute, a building plan that was carved into the stones only after the final mound, the third of the triple complex, was constructed!

The continuity of building design in Irish, Welsh and French Atlantic Megalithic structures indicates that the Star Temples of the New Stone Age were used as almanacs by any tribe with similar values, and were therefore very real computers. It should be stressed that these Computer Temples

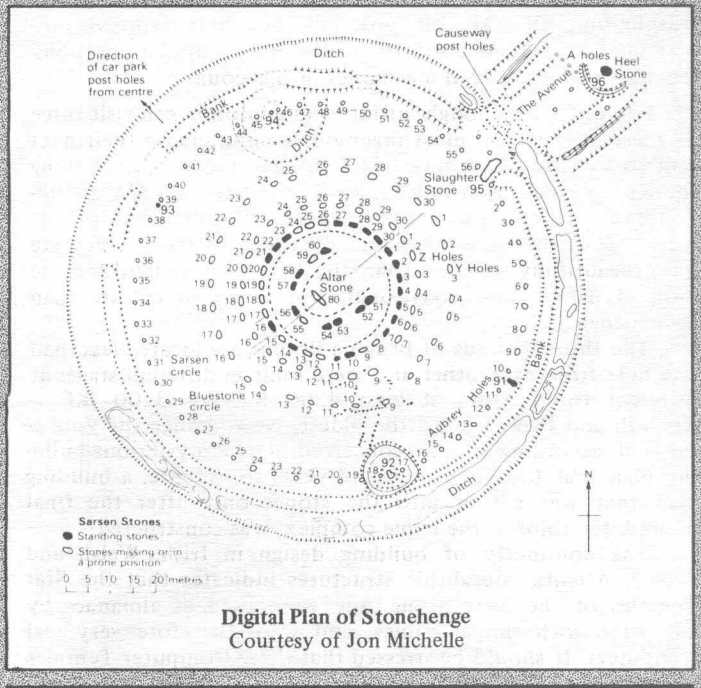
were not designed in the Bronze Age (between 3000 and 1800 B.C.) as is assumed by a number of poorly informed explorers, but rather on the horizon of settlement and agriculture. In fact, they mark the end of the Mesolithic era.

The question arises: What kind of society could have carried out such a continuity of thought and belief? They certainly could not have been savages in the strict sense of the word. Who then did build the Computer Temples of the pre-Celtic dawn and who brought them to North America?

There is a paradox in all of this. When we look at the mounds, the sun setting behind them, knowing that each December twenty-first a ray of light flashes into New Grange for approximately seventeen minutes and illuminates a triple spiral carving on the inner chamber wall, just as it has done for more than five thousand years, we realize that we are stranded between two great eras in human history. The old computer age operated by the gears of the Cosmos, and our hapless modern age with its electronic mainframe computers operated by fallible, unsure, oft-times illogical human beings, who have lost track of the cosmic gear system and have begun to rely on blind faith, prayer, and the not-so-infallible microchip. The entire spectrum of human achievement arcs between these two computer systems, creating a most amazing dilemma. It is obvious that we are meandering toward a point in evolution whereby electronic systems will be used to decipher the ancient stone computers, and yet these ancient mound computers probably hold the key to the secret vault of human existence, racial memories, our roots as human voyagers on this planet.

A crisis seems to arise when a given culture conforms to an unnatural, synthetic clocking system based on nonhuman and irrational tenets, or when a given culture falls away by reason of economic failure, famine, enslavement, or simple vanity, from old and well-tried natural rhythms. Again and again societies and individuals succeed, or fail, relative to their fulfillment or rejection of the massive natural computational skills latent in the human brain.

A computer, be it stone or electronic, is nothing more than an externalization of human, or mammalian, cognition. The stone computer must, by definition, be naturalistic as it requires light and shadow, water, earth and fire for its operation. An electronic computer, on the other hand, is easier to abuse: it can be removed from nature, can be synchronized to a corporate brain and can, as has been demonstrated in modern America, lead whole populations away from nature. There is, therefore, some validity to the anxieties of millions of people who fear modern electronic computers.



Digital Plan of Stonehenge
Courtesy of Jon Michelle

It is as if the entire evolution of our species has been dedicated to setting the stage for a series of single event horizons so huge, so sweeping, that they yet may change the course of human history irreversibly. The development of the microcomputer tool is one of these horizons.

No matter how fantastic predictions about the microcomputer may be, one thing remains certain and provable: Ancient societies were far more enlightened and regulated than we have ever realized and static archaeological theories are reeling from the shock of certain discoveries. Is it possible that the binary computer as we know it, functioning with light beams instead of electronic circuits, was developed more than five thousand years ago? Some scholars think so.

The proof of the computing abilities of certain Neolithic temples can be seen in a number of examples. Take, for instance, Stone No. 69 at Knowth, a mound which represents one third of the temple complex of Brug na Boinne, a mound which will be under excavation throughout the latter half of the twentieth century – a temple computer now in jeopardy of destruction because the head excavator believes it to be a tomb. Stone No. 69, referred to as "The Calendar Stone," weighs over twelve tons. The wavy line across the center calculates the lunar year. The spiral represents the sun.

Certain curvilinear markings are systematically displayed all over the stones, not at random as some early observers have remarked, but with great care. This purposefulness alone would be astonishing, but the round and semicircular markings are more than rural decorations; they are in fact *words*, words which act as markers for the cycles of the moon and sun. The markings of Stone No. 69 at Knowth exactly correlate with the phases of the moon. This stone is a *moon* computer. But others are directed by both sun and moon and/or stars and planets.

Each of the three mounds in the Boyne complex seem to have a slightly different orientation. Knowth and Dowth, in Irish Gaelic, are called *Gnobda* (that is, the light of knowledge) and *Dubhad* (the occlusion of light). Calibrated radio carbon dates establish Knowth and Dowth as having been constructed about 4200 B.C., or six thousand years ago. New Grange, a real upstart, seems to have been completed by 3500-3200 B.C., yet all three seem to operate interactively.

The original name of New Grange is controversial, but it, too, had an Irish name at one time, a name probably derived from the pre-Indo-European root word *grainne*, or "the sun." Thus, the names of the three mounds represent direct sunlight, indirect sunlight and shadow – and all three are at least fifteen hundred years older than Stonehenge, and some eighteen hundred years older than the Pyramids of Giza. Only the Rhomboid Pyramid overlaps Stonehenge, and only the *wooden* phase of Stonehenge overlaps the final stages of New Grange.

Of the three major Boyne Computer Temples – Knowth, Dowth, and New Grange – only New Grange, the youngest, has been restored to any degree. Knowth has been torn down by the Irish Government and will be rebuilt after a first or second century Christian Church has been dug up. Who knows what might happen to the entire structure when this is done. In fact Dowth is for sale. The only thing that saves any of the large computer mounds in the Atlantic sphere is their massive size – some of the stones used in their construction weigh twenty tons.

The ray of light observable at New Grange on December twenty-first each year lasts for exactly 17.3 minutes and seems to break through a cloud cover, even in rainy conditions, to reach its predetermined point as it has done for five thousand or more years. Why was this beam dial constructed? Was this a pointer, a calculator to mark a precise end or beginning of the year? What kind of civilization, except one

The machine-like codes so far discovered are binary, ternary and octal.

with a deep sense of scientific curiosity, could possibly need that degree of precision?

The concept of a beam dial was probably connected to a ritualized phenomenon of some kind. For the builders of New Grange and the other Passage Temples computing was religion. So what's new? But there is more going on here. After the seventeen-minute interval, the light beam makes its way back down the passageway, ceases to be a beam, leaves New Grange, and starts its apparent climb into the midday sky, describing an ogival arch identical to the shape of the beehive-like cavern inside New Grange. The sun then appears to set in the late afternoon, and finishes its day at Dowth -- the repository of shadow.

At Stonehenge the sun appears to rise over the Heel Stone on the dawn of June twenty-first, the summer solstice. It, too, is a well-known computer, established as such in 1965 by Dr. Gerald Hawkins, who used a computer to prove a computer. However, during the winter solstice at New Grange the sun appears to rise over a stone, and then enter a chamber to illuminate a marking, after being reflected from a huge basin stone situated in the center of the inner chamber.

Obviously there is more going on here than mere observation. The thing is counting! The computer mounds of the Atlantic Neolithic are operated by the apparent motion of the sun, moon and stars at different times of the year. Obviously these computers are not powered by electricity. These massive stone structures are driven by *light*, or rather by the dynamism between dark and light. Light, or the lack of it, can be seen as *binary*. Things can be either dark or light, on or off, yet with light a third twilight, or *ternary*, stage can exist, making things read +1, 0, -1. The passages in the computer temples are therefore designed to create a sharp binary contrast between light and dark. The light drives the computer, and the markings on the stones are illuminated at different times, depending upon the incident angle of light as it strikes the marking and moves across to the next stone. There are dozens of stones with hundreds of markings, and all are placed in very precise ways. These piles of rocks are computers.

Certain stones relate to the lunar calendar, others to planet orbits, others to sun cycles. Years, months, even centuries or eons could be accounted for in this manner. There are stones that are sundials and even moondials. There are vertical sundials and perpendicular sundials; there are water-dripping clocks built into stones, flat surfaces where skins may have been engraved with burn marks made by quartz magnifying glasses, birth chambers, death chambers, observation points, and much more.

The machine-like codes discovered so far are binary, ternary, and octal, but still other numeric systems may be found throughout the Atlantic Neolithic sphere. The most demonstrable of these "stone counting schema" is octal because there were eight major dates in the Atlantic Calendar year. This practice was retained by the Celts, who observed eight holidays: Imbolc, Vernal Equinox, Beltane, Summer Solstice, Lugnasa, Autumnal Equinox, Samhain and Winter Solstice. The transitions of the seasons are marked by the double and triple spirals found on the stones.

At Knowth, the sun enters the chamber to the east at vernal equinox, and sets in the west chamber after appearing to balance on a nine-foot obelisk. This effect is heightened every eighteen years when the sun appears to set as the full moon rises on the same obelisk; apparently this arrangement is also an eclipse predictor.

Throughout the entire Atlantic sphere there seems to be conformity of symbolic language carved into the stones. The conservative Irish have long thought that these ancient stone markings represented a form of language, but these topics were only whispered about in the most esoteric pubs.

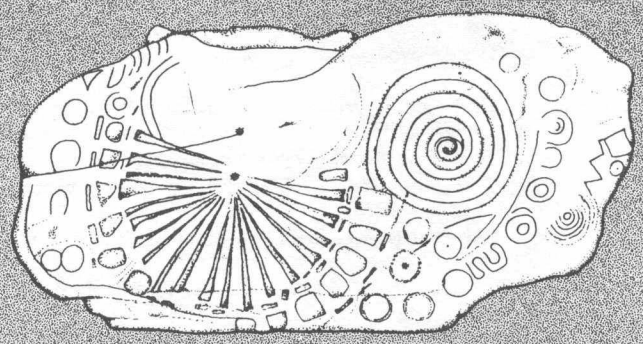
In any case, for whatever reason, the stone temples survived, until the 1960's when, for reasons connected to the Stonehenge discoveries, they began to be understood, and also plundered by amateur archaeologists and misguided professionals alike.

Two modern scholars came to their rescue, each working independently of the other -- two scholars who, between them, should probably have been given an award for their simple understanding of the ancient computer system. Dr. Elizabeth Shee-Twoig, an archaeologist from Cork, Eire, has spent more than twenty years compiling a complete survey of every major monument in the Atlantic sphere dating to the Megalithic Age. Her epic tome, *Megalithic Art in Western Europe*, published in 1982 by Oxford University Press and costing over $150, is three inches thick and profusely illustrated. This book is, to say the least, a gold mine of coded information for anyone interested in computer archaeology.

Martin Brennan, on the other hand, is an Aikido master and graphics illustrator from Brooklyn, New York, who was sent to Ireland by his Aikido master in Japan, a member of an ancient group known as the Japan Creation Group, specifically to teach Aikido and to study the stone temples of Ireland. Brennan knew ahead of time that they were sacred structures, but he had no idea they were computers until he arrived in the Boyne Valley.

Brennan had his insight into the codes of the Megaliths while exploring Dowth. It occurred to him that these stones and their carvings were interconnecting intaglio messages, instruction sets for lunar and solar calculations. He began sketching the entire Boyne complex and wound up spending ten years working in an advertising agency in Dublin to finance his extensive expeditions to Mexican, Irish, French and British sites. Both scholars are eminently qualified. Twoig's credentials are not in doubt, and although Brennan has no postgraduate degree, he certainly holds a degree in Art from Pratt Institute in New York and continues to be meticulous in his data collection techniques.

The fact remains that Brennan's data, regardless of method of collection, matches Twoig's data, with some minor exceptions. Both scholars claim the symbols in stone are meaningful. Both claim the symbols are art products as well as coded messages. Both agree that the carvings are positioned in certain static formal arrays like address registers, and both agree that the carvings output information when accessed by



Lunar/Solar Sundial -- Knowth, Ireland

sunlight and shadow. The codes are carved deeply into massive stones. These stones were purposefully put in place to express something larger than the carvings themselves. Brennan claims all the Megaliths in the Atlantic sphere are relational. Twoig views most of the Atlantic computers as separate entities. *Neither* point has been proved at this time, but certain similarities do exist.

From material collected by these scholars, and a growing number of other researchers, it is easy to speculate that the sun and moon were captured twenty-four hours each day and that the entire complex must be thought of as an observatory.

The markings on the Megaliths fall statistically into groups, the continental carvings being somewhat more eerie and perhaps more ancient. The Irish groupings are somewhat more geometric, although clearly both subsets are derived from an older, perhaps legendary, master root. A root Atlantic language which frankly we may never see first hand, except through these subsets.

The architects of the Neolithic era understood nature in a natural and yet scientific way and set about expressing their understanding in stone. It is easy to presume that these computers were left behind, like time capsules, for us to discover. But, this is vanity. More probably the stones were left for the grandchildren of the tribe, to assure a sense of cultural continuity, unless one assumes that "we" are the grandchildren.

For many reasons it seems prudent to view the entire wealth of human history, both known and unknown, actual and speculative, and the entire collection of human knowledge, including the lost knowledge now being retrieved, as data in the history of consciousness.

The process of the Neolithic Atlantic consciousness can be traced in the Gothic Cathedrals, the Pyramids, Medieval Alchemy, Machu Picchu and in many other fields, including modern technology. The beam of light at New Grange is expressed in modern terms as an edge of a large laser, and the language codes of the stone temples, which are also probably musically related to the octave, can be seen as dynamic elements in artificial intelligence programming. In other words, we are all on the threshold of a discovery so vast that Space will be of passing interest. We are about to discover the Human Being.

■■

Bibliography

1. Ardry, Robert. *African Genesis*. Morrow, New York, 1964.
2. Brennan, Martin. *The Boyne Valley Vision*. Dolmen Books, Dublin, 1980.
3. Brennan, Martin. *The Stars and the Stones*. Thames and Hudson, London, (in press) 1983.
4. Campbell, Joseph. *The Flight of the Wild Gander*. Pantheon, New York, 1963.
5. Fell, Barry. *Bronze Age America*. Little Brown, Boston, 1982.
6. Hawkins, Gerald. *Stonehenge Decoded*. Various editions, 1965 F.
7. Herity, M. and Eogan, G. *Ireland in Pre-History*. Routlidge & Kegan Paul, London, 1977.
8. Marshak, Alexander. *National Geographic*. May, 1975.
9. Michell, John. *A Little History of Astro-Archaeology*. Thames and Hudson, London, 1978.
10. O'Kelly, Clare. *Newgrange*. Irish Government Publication, 1969.
11. Shee-Twoig, Elizabeth. *Megalithic Art in Western Europe*. Oxford, 1981.
12. Tomkins, Peter. *The Magic of Obelisks*. Harper and Row, 1981.

Software Review

JRT Pascal: Another Look

by Daniel S. Hunt

JRT Pascal is not quite as bad as Dave Cortesi claims and deserves a spirited defense. It is possible that Cortesi failed to read the manual carefully, and did not make the effort to struggle through the admittedly tough first few lines of original code. Instead, he took the easy way out and savaged the product on the dubious point of language standard and portability, which, frankly, no commercial Pascal has.

I submit that JRT Pascal is not only a bargain, but an absolute steal. I enjoyed its good fit to CP/M and departure from some of the more irksome features of standard Pascal so much that I sent my money in for my own disk and manual after I'd received a pirated copy.

JRT seems to meet business programming needs better than a Jenson and Wirth standard Pascal can. Due to the CALL function and SYSTEM functions, it works in harmony with CP/M and fully makes use of its facilities.

The real standard is CP/M, determined not by theory but by market need. Where CP/M goes, so goes JRT. That will be plenty for most programmers. An additional benefit of JRT is that royalties are not charged for the run-time EXEC needed to run intermediate code. This means that you may distribute the EXEC free of charge to your customers along with your compiled JRT INT files.

The declaration of a character variable as "string[1]" would have allowed Cortesi to sample single characters without carriage returns. Granted this is not standard. But consider the alternatives the language processors offer – Fortran-80 and PL/I-80 coming to mind – where a read-character routine must be written in assembly language and linked in each compile-time cycle. I wrote a test program that uses the string[1] declaration as well as the concatenation features so quickly dismissed by Cortesi to produce a SuperCalc-style data entry module. It went together quickly and worked well – the bottom line on any language. I also wrote JRT Pascal programs that successfully used sequential character and line delimited I/O, and fixed record sequential and random I/O (with no padding of odd-length records). The classic Wirth recursive quicksort was implemented quickly from the book *Algorithms + Data Structures = Programs*.

JRT has no exclusive on compiler failure. Try Pascal M/T, or UCSD Pascal, or sample Digital's PL/I-80 version 1.3, or Microsoft's Fortran compiler. These latter two occasionally blow up if a mispunctuation distorts block structure. Digital's PL/I-80 will on occasion deliver ten error messages for only one error. Mike Lehman's Pascal MT-Plus in its initial Version 5 costing $325.00 plus $100.00 for each upgrade had sequential file I/O problems and inconsistently executed console I/O from one version of the read statement to the next. Are any of the above compilers "bad"? I think not. So what do you do when these bugs show? You learn to type the code in right, a process which Cortesi states he didn't take the trouble to do! Bottom line: JRT's compiler is average on error handling, but better than average on verbosity when it catches the right error.

JRT's string handling and less-than-rigid typing are extremely useful in business programming, particularly in the production of standard-length fields, and formatting. I particularly like the ability to use the statement "write(astring:32)" and have the ability to set a string of characters in a fixed-length field. I also find the ability to assume the occurrence of truncation or padding of unequal length structures in assignment a great time saver. The very popular data base language, DBASEII, does this, allowing flexibility in resizing and conversion of existing data bases and nobody is complaining. So why should it be a point of criticism for JRT Pascal? Using yet another example, MBASIC and CBASIC are great commercial successes because they get the job done. So do we criticize them for being non-standard BASIC? Nope. We make use of them, moaning and bitching all the while but cranking out fabulous things, as demonstrated in prior issues of *Dr. Dobb's Journal*.

Why should you have to put up with conceptually elegant file I/O concepts that sprang forth in the days when tape was the leading form of mass storage, just to adhere to the Pascal standard? I frankly get tired of writing hundreds of lines of library code in "standard" Pascal to produce commonly used forms of file access. One of JRT's few shortcomings in this area is that it failed to include a non-standard equivalent to BASIC's line input statement.

A better starting point for Cortesi's critique would have been a comparison to the CBASIC compiler/interpreter. Both CBASIC and JRT Pascal have similar intent. They are languages oriented to business programming. CBASIC leads in age and maturity and thus has fewer bugs in it. CBASIC's compilation speed is about 2 to 2.5 times faster than JRT's. In turn, JRT's compilations run 2 to 2.5 times faster than those in PL/I-80 or Pascal/MT.

JRT Pascal beats CBASIC in both integer and BCD real math processing speed and flexibility of data structures. It interfaces to the CP/M system and assembly language modules more easily than CBASIC. My benchmark tests of JRT against BASIC-80 in double-precision mode and PL/I-80 (a native code compiler) in fixed decimal mode give JRT a small edge in addition and multiplication and only a 1.5-to-1 disadvantage against PL/I-80 for division. JRT's results, by the way, are decimally correct. Like CBASIC, JRT Pascal uses 14-digit precision floating decimal math.

CBASIC's intermediate files, given an identical program, seem to be a tad smaller than JRT's. JRT's dynamic overlay manager and storage recovery techniques give it the edge in program modularity, allowing separate intermediate files for generalized subroutines as well as major subprogram modules. Either language allows you to create massive, complex systems of subprograms that use a minimum of disk space.

On the other hand, the bugs in JRT are real, but you can write around them. If you're good. It's not a start-up tool, but I believe Cortesi overreacted. Incidentally, I found a number of bugs nobody has mentioned as yet.

I couldn't get the %NOLIST compiler directive to work, so it listed code on the console as it compiled (which probably slows it down somewhat). JRT wouldn't read record zero in random I/O mode, using the RRN (relative record number) form of random access. The latter problem was written around by using JRT's unique RBA (relative byte address) random I/O feature to access record 0. RRN and RBA may be used interchangeably without closing and reopening files, which leads to marvelous possibilities for constructing files with built-in indexes. Tyson is aware of the RRN = 0 problem and is working on it.

Some of the problems that occurred in initial shipments of JRT packages resulted from the fact that INT files compiled with the Version I compiler were inadvertently placed on the Version II distribution disks. Version I files are not compatible with the Version II executive. Tyson, to my knowledge, has been making good on the bad disks that slipped out. My first copy of JRT was "pirated," which is okay with Tyson because I received it as a gift. Liking it very much, I sent in the $29 for my own copy and received a disk with the correct INT files, which I pirated right back to the guy who gave me the original pirated copy.

JRT's implied claim that all components of his system will run in 56K is not entirely true. It's true of the compiler itself, but not true of JRTASM, an assembler that creates either INT or COM files. Fortunately, Tyson has included a program called CONVERTM that will convert assembled Microsoft or RMAC REL files into INT files that may be run as callable procedures. One of the subroutines called by CONVERTM, testbit.asm, does not appear on the distribution disk as an INT file. So, you can't use CONVERTM either. My solution was to find a full 64K machine and use JRTASM to assemble testbit.asm into an intermediate file. As the output is only 1 CP/M sector long, I dumped it with DDT, took the hardcopy home, patched it back into DDT, and then saved it as TESTBIT. INT. Now CONVERTM runs and I can use relocatable REL files in my 58K system. It's a long way around, and I hope Tyson still reads *Dobb's*.

PS

After I wrote the above critique I ran a critical pass program. The FOR INDEX= 1-4 statement hung after the second iteration of output to screen and instead sent the output to the printer, which was switched on at the time. This is another potentially real problem for Tyson and he should really get a handle on it soon as the output not only goes to the printer but takes the keyboard over there too. Somehow I/O byte location 03 hex got modified to 01 hex. The program is not error free yet, but at 30 bucks I, for one, ain't complaining. I do, on the other hand, sympathize with those who have had trouble with any of these problems . . . someday they will be ironed out.

Rebuttal

by D.E. Cortesi

Dan Hunt seems to be making two main points: (1) language standards aren't important, other compilers violate the Pascal standard, and anyway JRT's nonstandard features are useful, so they shouldn't be criticized; and (2) JRT Pascal has lots of bugs, but so do other products and anyway a good programmer can code around most of them. I don't buy either of these propositions, and I'd like to explain why.

I don't think I have to work very hard to defend the idea of language standards. Think about the difficulty of moving even a simple BASIC program from a TRS-80 to an Atari. Compare that to the ease of moving a FIG-FORTH program between *any* two machines. The benefits of an effective standard are obvious. If Hunt really doesn't understand this now, he will begin to appreciate it when he decides to upgrade to a 16-bit system. If he is lucky, JRT Systems will have a new compiler for the system he wants to move to (if he's *very* lucky, it will be compatible with their old one). If they haven't, Hunt will find himself rewriting all his neat programs. He won't have looked at them in months; the wasted effort and time will be quite painful. And if he has written programs for pay, what are his customers or employers to do when *they* want to upgrade?

I noted in my review that Pascal implementors are often forced to extend the language. There is a big difference between *extending* Pascal (with new predeclared types and functions) and *altering* it with gratuitous twiddles of the core language as defined by Wirth (and by ISO). The difference is this: so long as the core is implemented correctly, a program that uses only those standard features can be ported to any compiler that also implements the core language. That includes many systems besides 8-bit CP/M. CP/M is definitely *not* "the standard"; a program in standard Pascal can run on UNIX and TOPS and MSDOS, on 8086s and 6800s and 370s and VAXes. I did not criticize JRT Pascal for its extensions (many of which are useful), but for its completely unnecessary alterations of the core language.

Now, as to the product's bugginess. Nothing in JRT's ads or manual indicated that it was being sold for hackers only. It is sold as a finished, professional product, ready for use by anyone who knows Pascal. That's the standard against which I measured it, with the results I described. I wasn't reviewing any other compiler, I was reviewing this one. I reported the large number of compiler crashes that turned up, a much larger number than I could ever tolerate in an important software tool.

I will accept Hunt's assertion that I could have coded around the problems. But I must ask, *why should I have to*? Maybe Hunt feels that when a compiler crashes, it has presented him with a fun riddle, a pleasant chance to exercise his wits. To me, a compiler crash is an infuriating interruption in the flow of my work. To Hunt, there's nothing wrong with having to recode and recompile, maybe several times, in order to coax the compiler into doing its job. To me, that sort of thing is a total waste of my time, which is all too limited as it is.

But okay, I will revise my conclusions. If you write programs only for your own use, *and* if you expect never, ever, to move to different hardware, *and* if you enjoy puzzling out and coding around compiler failures, then you will find JRT Pascal an acceptable product. If you, like me, want the ability to write (or use) portable code, or if you believe as I do that a compiler should compile, regularly and reliably, then you will find JRT Pascal Not Acceptable just as I did.

Small-C Compiler, v.2

Listing One (Part four continued)

The following listing is a continuation of the Small-C Compiler featured last month. A detailed explanation and abstract can be found beginning on page 16 of *DDJ* No. 74. Suffice it to say this program example proves the *Dobb's* philosophy works as it has drawn from dozens of articles that have appeared in *DDJ* since Ron Cain's original in *DDJ* No. 45.

```
160   char c;
161   if (match(quote)==0) return 0;
162   val[0]=litptr;
163   while (ch!='"') {
164     if(ch==0) break;
165     stowlit(litchar(), 1);
166     }
167   gch();
168   litq[litptr++]=0;
169   return 1;
170   }
171
172 stowlit(value, size) int value, size; {
173   if((litptr+size) >= LITMAX) {
174     error("literal queue overflow"); abort(ERRCODE);
175     }
176   putint(value, litq+litptr, size);
177   litptr=litptr+size;
178   }
179
180 /*
181 ** return current literal char & bump lptr
182 */
183 litchar() {
184   int i, oct;
185   if((ch!=92)|(nch==0)) return gch();
186   gch();
187   if(ch=='n') {gch(); return 13;} /* CR */
188   if(ch=='t') {gch(); return  9;} /* HT */
189   if(ch=='b') {gch(); return  8;} /* BS */
190   if(ch=='f') {gch(); return 12;} /* FF */
191   i=3; oct=0;
192   while(((i--)>0)&(ch>='0')&(ch<='7')) oct=(oct<<3)+gch()-'0';
193   if(i==2) return gch(); else return oct;
194   }
195
```

(Part five)

```
 1 /*
 2 ** Small-C Compiler Version 2.0
 3 **
 4 ** Copyright 1982 J. E. Hendrix
 5 **
 6 ** Part 4
 7 */
 8 #include stdio.h,2
 9 #include cc.def,2
10
11 /*
12 ** external references in part 1
13 */
14 extern char
15 #ifdef HASH
16  *macn,
17 #endif
18 #ifdef OPTIMIZE
19  optimize,
20 #endif
21  *stagenext, ssname[NAMESIZE];
22 extern int
23   beglab, csp, output;
24
25 /*
26 ** external references in part 2
27 */
28 extern int
29 #ifdef HASH
30   search(),
31 #else
32   findmac(),
33 #endif
34   clearstage(), col(), cout(), getint(), getlabel(),
35   nl(), numeric(), ol(), ot(), printlabel(),
36   lout(), outdec(), outstr(), streq();
37
38 /*
39 ** external references in part 3
40 */
41 extern int const();
42
43 #include cc41.c,2
44 #include cc42.c,2
```

(Continued on column 2)

```
 1 /*
 2 ** print all assembler info before any code is generated
 3 */
 4 header()  {
 5   beglab=getlabel();
 6 #ifndef LINK
 7   if(beglab < 3)
 8 #endif
 9     {
10 #ifdef SMALL_VM
11     ol("LXI D,$+6");
12     ol("JMP CC9998");
13 #endif
14     jump(beglab);
15     }
16   }
17
18 /*
19 ** print any assembler stuff needed at the end
20 */
21 trailer()  {
22 #ifdef SMALL_VM
23 #ifndef LINK
24   if((beglab == 1)|(beglab > 9000))
25 #endif
26     {
27     ol("CC9997:JMP CCB0J");
28     ol("CC9998:DS 6");
29     ol("PUSH D");
30     ol("LXI D,$+6");
31     ol("JMP CC9997");
32     ol("ORG CC9998");
33     ol("JMP $+6");
34     }
35 #endif
```

(Continued on column 1, page 50)

Small-C Compiler, v.2

Listing One (Part five continued)

```
36    ol("END");
37    }
38
39  /*
40  ** load # args before function call
41  */
42  loadargc(val) int val; {
43  #ifdef HASH
44    if(search("NOCCARGC", macn, NAMESIZE+2, MACNEND, MACNBR, 0)==0) {
45  #else
46    if(findmac("NOCCARGC")==0) {
47  #endif
48      ot("MVI A,");
49      outdec(val);
50      nl();
51      }
52    }
53
54  /*
55  ** declare entry point
56  */
57  entry() {
58    outstr(ssname);
59    col();
60  #ifdef LINK
61    ol("ENTRY");
62  #else
63    nl();
64  #endif
65    }
66
67  /*
68  ** declare external reference
69  */
70  external(name) char *name; {
71  #ifdef LINK
72    outstr(name);
73    col();
74    ol("EXTRN");
75  #endif
76    }
77
78  /*
79  ** fetch object indirect to primary register
80  */
81  indirect(lval) int lval[]; {
82    if(lval[1]==CCHAR) call("CCGCHAR");
83    else               call("CCGINT");
84    }
85
86  /*
87  ** fetch a static memory cell into primary register
88  */
89  getmem(lval)  int lval[]; {
90    char *sym;
91    sym=lval[0];
92    if((sym[IDENT]!=POINTER)&(sym[TYPE]==CCHAR)) {
93      ot("LDA ");
94      outstr(sym+NAME);
95      nl();
96      call("CCSXT");
97      }
98    else {
```

```
164 ** into secondary register
165 */
166 immed2() {
167   ot("LXI D,");
168   }
169
170 /*
171 ** push primary register onto stack
172 */
173 push() {
174   ol("PUSH H");
175   csp=csp-BPW;
176   }
177
178 /*
179 ** unpush or pop as required
180 */
181 smartpop(lval, start) int lval[]; char *start; {
182   if(lval[5])  pop(); /* secondary was used */
183   else unpush(start);
184   }
185
186 /*
187 ** replace a push with a swap
188 */
189 unpush(dest) char *dest; {
190   int i;
191   char *sour;
192 #ifdef TAB
193   sour="\tXCHG;:";  /* peephole() uses trailing ";:" */
194 #else
195   sour="XCHG;:";  /* peephole() uses trailing ";:" */
196 #endif
197   while(*sour) *dest++ = *sour++;
198   sour=stagenext;
199   while(--sour > dest) { /* adjust stack references */
200 #ifdef TAB
201     if(streq(sour,"\tDAD SP")) {
202 #else
203     if(streq(sour,"DAD SP")) {
204 #endif
205       --sour;
206       i=BPW;
207       while(numeric(*(--sour))) {
208         if((*sour=*sour-i) < '0') {
209           *sour=*sour+10;
210           i=1;
211           }
212         else i=0;
213         }
214       }
215     }
216   csp=csp+BPW;
217   }
218
219 /*
220 ** pop stack to the secondary register
221 */
222 pop() {
223   ol("POP D");
224   csp=csp+BPW;
225   }
226
227 /*
228 ** swap primary register and stack
229 */
230 swapstk() {
231   ol("XTHL");
232   }
```

```
 99      ot("LHLD ");
100     outstr(sym+NAME);
101     nl();
102     }
103   }
104
105 /*
106 ** fetch addr of the specified symbol into primary register
107 */
108 getloc(sym)  char *sym; {
109   const(getint(sym+OFFSET, OFFSIZE)-csp);
110   ol("DAD SP");
111   }
112
113 /*
114 ** store primary register into static cell
115 */
116 putmem(lval)  int lval[]; {
117   char *sym;
118   sym=lval[0];
119   if((sym[IDENT]!=POINTER)&(sym[TYPE]==CCHAR)) {
120     ol("MOV A,L");
121     ot("STA ");
122     }
123   else ot("SHLD ");
124   outstr(sym+NAME);
125   nl();
126   }
127
128 /*
129 ** put on the stack the type object in primary register
130 */
131 putstk(lval) int lval[]; {
132   if(lval[1]==CCHAR) {
133     ol("MOV A,L");
134     ol("STAX D");
135     }
136   else call("CCPINT");
137   }
138
139 /*
140 ** move primary register to secondary
141 */
142 move() {
143   ol("MOV D,H");
144   ol("MOV E,L");
145   }
146
147 /*
148 ** swap primary and secondary registers
149 */
150 swap() {
151   ol("XCHG;;");  /* peephole() uses trailing ";;" */
152   }
153
154 /*
155 ** partial instruction to get immediate value
156 ** into the primary register
157 */
158 immed() {
159   ot("LXI H,");
160   }
161
162 /*
163 ** partial instruction to get immediate operand
```

(Continued on top of page 50)

```
233
234 /*
235 ** process switch statement
236 */
237 sw() {
238   call("CCSWITCH");
239   }
240
241 /*
242 ** call specified subroutine name
243 */
244 call(sname)  char *sname; {
245   ot("CALL ");
246   outstr(sname);
247   nl();
248   }
249
250 /*
251 ** return from subroutine
252 */
253 ret() {
254   ol("RET");
255   }
256
257 /*
258 ** perform subroutine call to value on stack
259 */
260 callstk() {
261   immed();
262   outstr("$+5");
263   nl();
264   swapstk();
265   ol("PCHL");
266   csp=csp+BPW;
267   }
268
269 /*
270 ** jump to internal label number
271 */
272 jump(label)  int label; {
273   ot("JMP ");
274   printlabel(label);
275   nl();
276   }
277
278 /*
279 ** test primary register and jump if false
280 */
281 testjump(label)  int label; {
282   ol("MOV A,H");
283   ol("ORA L");
284   ot("JZ ");
285   printlabel(label);
286   nl();
287   }
288
289 /*
290 ** test primary register against zero and jump if false
291 */
292 zerojump(oper, label, lval) int oper, label, lval[]; {
293   clearstage(lval[7], 0);  /* purge conventional code */
294   oper(label);
295   }
296
297 /*
```

(Continued on next page)

Small-C Compiler, v.2

Listing One (Part five continued)

```
298 ** define storage according to size
299 */
300 defstorage(size) int size; {
301   if(size==1) ot("DB ");
302   else        ot("DW ");
303   }
304
305 /*
306 ** point to following object(s)
307 */
308 point() {
309   ol("DW $+2");
310   }
311
312 /*
313 ** modify stack pointer to value given
314 */
315 modstk(newsp, save)  int newsp, save; {
316   int k;
317   k=newsp-csp;
318   if(k==0)return newsp;
319   if(k>=0) {
320     if(k<7) {
321       if(k&1) {
322         ol("INX SP");
323         k--;
324         }
325       while(k) {
326         ol("POP B");
327         k=k-BPW;
328         }
329       return newsp;
330       }
331     }
332   if(k<0) {
333     if(k>-7) {
334       if(k&1) {
335         ol("DCX SP");
336         k++;
337         }
338       while(k) {
339         ol("PUSH B");
340         k=k+BPW;
341         }
342       return newsp;
343       }
344     }
345   if(save) swap();
346   const(k);
347   ol("DAD SP");
348   ol("SPHL");
349   if(save) swap();
350   return newsp;
351   }
352
353 /*
354 ** double primary register
355 */
356 doublereg() {ol("DAD H");}
357
```

```
67 */
68 neg() {call("CCNEG");}
69
70 /*
71 ** one's complement primary register
72 */
73 com() {call("CCCOM");}
74
75 /*
76 ** increment primary register by one object of whatever size
77 */
78 inc(n) int n; {
79   while(1) {
80     ol("INX H");
81     if(--n < 1) break;
82     }
83   }
84
85 /*
86 ** decrement primary register by one object of whatever size
87 */
88 dec(n) int n; {
89   while(1) {
90     ol("DCX H");
91     if(--n < 1) break;
92     }
93   }
94
95 /*
96 ** test for equal to
97 */
98 eq()  {call("CCEQ");}
99
100 /*
101 ** test for equal to zero
102 */
103 eq0(label) int label; {
104   ol("MOV A,H");
105   ol("ORA L");
106   ot("JNZ ");
107   printlabel(label);
108   nl();
109   }
110
111 /*
112 ** test for not equal to
113 */
114 ne()  {call("CCNE");}
115
116 /*
117 ** test for not equal to zero
118 */
119 ne0(label) int label; {
120   ol("MOV A,H");
121   ol("ORA L");
122   ot("JZ ");
123   printlabel(label);
124   nl();
125   }
126
127 /*
128 ** test for less than (signed)
129 */
```

```
/*
** add primary and secondary registers (result in primary)
*/
add() (ol("DAD D");)

/*
** subtract primary from secondary register (result in primary)
*/
sub() (call("CCSUB");)

/*
** multiply primary and secondary registers (result in primary)
*/
mult() (call("CCMULT");)

/*
** divide secondary by primary register
** (quotient in primary, remainder in secondary)
*/
div() (call("CCDIV");)

/*
** remainder of secondary/primary
** (remainder in primary, quotient in secondary)
*/
mod() (div();swap();)

/*
** inclusive "or" primary and secondary registers
** (result in primary)
*/
or() (call("CCOR");)

/*
** exclusive "or" the primary and secondary registers
** (result in primary)
*/
xor() (call("CCXOR");)

/*
** "and" primary and secondary registers
** (result in primary)
*/
and() (call("CCAND");)

/*
** logical negation of primary register
*/
lneg() (call("CCLNEG");)

/*
** arithmetic shift right secondary register
** number of bits given in primary register
** (result in primary)
*/
asr() (call("CCASR");)

/*
** arithmetic shift left secondary register
** number of bits given in primary register
** (result in primary)
*/
asl() (call("CCASL");)

/*
** two's complement primary register
```

(Continued on page 52, column 2)

```
lt()  (call("CCLT");)

/*
** test for less than to zero
*/
lt0(label) int label; {
  ol("XRA A");
  ol("ORA H");
  ot("JP ");
  printlabel(label);
  nl();
  }

/*
** test for less than or equal to (signed)
*/
le()  (call("CCLE");)

/*
** test for less than or equal to zero
*/
le0(label) int label; {
  ol("MOV A,H");
  ol("ORA L");
  ol("JZ $+8");
  ol("XRA A");
  ol("ORA H");
  ot("JP ");
  printlabel(label);
  nl();
  }

/*
** test for greater than (signed)
*/
gt()  (call("CCGT");)

/*
** test for greater than to zero
*/
gt0(label) int label; {
  ol("XRA A");
  ol("ORA H");
  ot("JM ");
  printlabel(label);
  nl();
  ol("ORA L");
  ot("JZ ");
  printlabel(label);
  nl();
  }

/*
** test for greater than or equal to (signed)
*/
ge()  (call("CCGE");)

/*
** test for gteater than or equal to zero
*/
ge0(label) int label; {
  ol("XRA A");
  ol("ORA H");
  ot("JM ");
  printlabel(label);
```

(Continued on next page)

Small-C Compiler, v.2

Listing One (Part five continued)

```
195   nl();
196   }
197
198 /*
199 ** test for less than (unsigned)
200 */
201 ult()  (call("CCULT");)
202
203 /*
204 ** test for less than to zero (unsigned)
205 */
206 ult0(label) int label; {
207   ot("JMP ");
208   printlabel(label);
209   nl();
210   }
211
212 /*
213 ** test for less than or equal to (unsigned)
214 */
215 ule()  (call("CCULE");)
216
217 /*
218 ** test for greater than (unsigned)
219 */
220 ugt()  (call("CCUGT");)
221
222 /*
223 ** test for greater than or equal to (unsigned)
224 */
225 uge()  (call("CCUGE");)
226
227 #ifdef OPTIMIZE
228 peephole(ptr) char *ptr; {
229   while(*ptr) {
230 #ifdef TAB
231     if(streq(ptr, "\tLXI H,0\n\tDAD SP\n\tCALL CCGINT")) {
232       if(streq(ptr+31, "XCHG\;")) (pp2()\;ptr=ptr+38\;)
233       else                        (pp1()\;ptr=ptr+30\;)
234       }
235     else if(streq(ptr, "\tLXI H,2\n\tDAD SP\n\tCALL CCGINT")) {
236       if(streq(ptr+31, "XCHG\;")) (pp3(pp2)\;ptr=ptr+38\;)
237       else                        (pp3(pp1)\;ptr=ptr+30\;)
238       }
239     else if(optimize) {
240       if(streq(ptr, "\tDAD SP\n\tCALL CCGINT")) {
241         ol("CALL CCDSGI");
242         ptr=ptr+21;
243         }
244       else if(streq(ptr, "\tDAD D\n\tCALL CCGINT")) {
245         ol("CALL CCDDGI");
246         ptr=ptr+20;
247         }
248       else if(streq(ptr, "\tDAD SP\n\tCALL CCGCHAR")) {
249         ol("CALL CCDSGC");
250         ptr=ptr+22;
251         }
252       else if(streq(ptr, "\tDAD D\n\tCALL CCGCHAR")) {
253         ol("CALL CCDDGC");
254         ptr=ptr+21;
255         }
256       else if(streq(ptr,
```

```
325 "DAD SP\nMOV D,H\nMOV E,L\nCALL CCGINT\nDCX H\nCALL CCPINT")) {
326         ol("CALL CCDECI");
327         ptr=ptr+53;
328         }
329       else if(streq(ptr,
330 "DAD SP\nMOV D,H\nMOV E,L\nCALL CCGCHAR\nINX H\nMOV A,L\nSTAX D")) {
331         ol("CALL CCINCC");
332         ptr=ptr+57;
333         }
334       else if(streq(ptr,
335 "DAD SP\nMOV D,H\nMOV E,L\nCALL CCGCHAR\nDCX H\nMOV A,L\nSTAX D")) {
336         ol("CALL CCDECC");
337         ptr=ptr+57;
338         }
339       else if(streq(ptr, "DAD D\nPOP D\nCALL CCPINT")) {
340         ol("CALL CCDDPDPI");
341         ptr=ptr+24;
342         }
343       else if(streq(ptr, "DAD D\nPOP D\nMOV A,L\nSTAX D")) {
344         ol("CALL CCDDPDPC");
345         ptr=ptr+27;
346         }
347       else if(streq(ptr, "POP D\nCALL CCPINT")) {
348         ol("CALL CCPDPI");
349         ptr=ptr+18;
350         }
351       else if(streq(ptr, "POP D\nMOV A,L\nSTAX D")) {
352         ol("CALL CCPDPC");
353         ptr=ptr+21;
354         }
355       /* additional optimizing logic goes here */
356 #endif
357       else cout(*ptr++, output);
358       }
359     else cout(*ptr++, output);
360     }
361   }
362
363 pp1() {
364   ol("POP H");
365   ol("PUSH H");
366   }
367
368 pp2() {
369   ol("POP D");
370   ol("PUSH D");
371   }
372
373 pp3(pp) int pp; {
374   ol("POP B");
375   pp();
376   ol("PUSH B");
377   }
378
379 #endif
```

Listing Two

```
1 /*
2 ** stdio.c -- standard I/O library
3 **
4 ** Copyright 1982 J. E. Hendrix
5 */
6 #include call.a
```

```
"\tDAD SP\n\tMOV D,H\n\tMOV E,L\n\tCALL CCGINT\n\tINX H\n\tCALL CCPINT")) {
        ol("CALL CCINCI");
        ptr=ptr+59;
        }
      else if(streq(ptr,
"\tDAD SP\n\tMOV D,H\n\tMOV E,L\n\tCALL CCGINT\n\tDCX H\n\tCALL CCPINT")) {
        ol("CALL CCDECI");
        ptr=ptr+59;
        }
      else if(streq(ptr,
"\tDAD SP\n\tMOV D,H\n\tMOV E,L\n\tCALL CCGCHAR\n\tINX H\n\tMOV A,L\n\tSTAX D")) {
        ol("CALL CCINCC");
        ptr=ptr+64;
        }
      else if(streq(ptr,
"\tDAD SP\n\tMOV D,H\n\tMOV E,L\n\tCALL CCGCHAR\n\tDCX H\n\tMOV A,L\n\tSTAX D")) {
        ol("CALL CCDECC");
        ptr=ptr+64;
        }
      else if(streq(ptr, "\tDAD D\n\tPOP D\n\tCALL CCPINT")) {
        ol("CALL CCDDPDPI");
        ptr=ptr+27;
        }
      else if(streq(ptr, "\tDAD D\n\tPOP D\n\tMOV A,L\n\tSTAX D")) {
        ol("CALL CCDDPDPC");
        ptr=ptr+31;
        }
      else if(streq(ptr, "\tPOP D\n\tCALL CCPINT")) {
        ol("CALL CCPDPI");
        ptr=ptr+20;
        }
      else if(streq(ptr, "\tPOP D\n\tMOV A,L\n\tSTAX D")) {
        ol("CALL CCPDPC");
        ptr=ptr+24;
        }
      /* additional optimizing logic goes here */
#else
    if(streq(ptr,"LXI H,0\nDAD SP\nCALL CCGINT")) {
      if(streq(ptr+27, "XCHG\n")) {pp2();ptr=ptr+34;}
      else                        {pp1();ptr=ptr+27;}
      }
    else if(streq(ptr,"LXI H,2\nDAD SP\nCALL CCGINT")) {
      if(streq(ptr+27, "XCHG\n")) {pp3(pp2);ptr=ptr+34;}
      else                        {pp3(pp1);ptr=ptr+27;}
      }
    else if(optimize) {
      if(streq(ptr, "DAD SP\nCALL CCGINT")) {
        ol("CALL CCDSGI");
        ptr=ptr+19;
        }
      else if(streq(ptr, "DAD D\nCALL CCGINT")) {
        ol("CALL CCDDGI");
        ptr=ptr+18;
        }
      else if(streq(ptr, "DAD SP\nCALL CCGCHAR")) {
        ol("CALL CCDSGC");
        ptr=ptr+20;
          }
      else if(streq(ptr, "DAD D\nCALL CCGCHAR")) {
        ol("CALL CCDDGC");
        ptr=ptr+19;
        }
      else if(streq(ptr,
"DAD SP\nMOV D,H\nMOV E,L\nCALL CCGINT\nINX H\nCALL CCPINT")) {
        ol("CALL CCINCI");
        ptr=ptr+53;
        }
      else if(streq(ptr,
```

(Continued on page 55, column 2)

```
#define NOCCARGC /* don't pass arg count to functions */
                 /* called by these functions */
#define SP 32
#define CTL_D 4
#define DEL 127
#define LST_PORT 9
#define LST_FILE 17
#define stdin 0
#define stdout 1
#define stderr 2
#define stdport 3
#define stdlist 4
#define ERR -2
#define EOF -1
#define NULL 0
#define CR 13
#define LF 10
#asm
GETC:ENTRY
JMP FGETC
PUTC:ENTRY
JMP FPUTC
FFLUSH:ENTRY
JMP CCFLUSH
ABORT:ENTRY
JMP CCEXIT
EXIT:ENTRY
JMP CCEXIT
UNLINK:ENTRY
JMP CCPURGE
#endasm

fgetc(fd)  int fd; {
  char c; int flag;
  while(1) {
    flag=CCREAD(fd, &c, 1);
    if(c!=LF) break;
    }
  if((flag==NULL);(flag==ERR)
    ;(c<=NULL)  /** OFFH is caught here **/
    ;(c==CTL_D)) return EOF;
  if(CCFILE(fd)==0) {
    if(c==DEL) c='\b';
    if(c!='\b') {
      if(fd==stdin) fd=stderr;
      fputc(c, fd);
      }
    }
  return(c&255);
  }

getchar()
  {return(fgetc(stdin));}

fgets(string, size, fd)  char string[]; int size, fd; {
  int k, sz, echo;
  char c;
  if(CCFILE(fd)!=0) {
    CCDELIM('\n');
    if((k=CCREAD(fd, string, --size)) < 1) return NULL;
    if(*string==LF) sz=1; else sz=0;
    if(string[sz]<=NULL) return NULL;  /* OFFH is caught **/
    if(string[--k]!='\n') ++k;
    string[k]=NULL;
    k=-1;
```

(Continued on next page)

Small-C Compiler, v.2

Listing Two (Part five continued)

```
72      while(string[++k]=string[sz++]);
73      }
74    else {
75      k=0; sz=size;
76      while((--sz>0)&((c=string[k]=fgetc(fd))!=CR)) {
77        if(c==EOF) return NULL;
78        if(c=='\b') {
79          if(k>0) {
80            --k;
81            ++sz;
82            if((echo=fd)==stdin) echo=stderr;
83            fputs("\b \b", echo);
84            }
85          }
86        else ++k;
87        }
88      string[k]=0;
89      }
90    return string;
91    }
92
93  fputc(c,fd)  char c;int fd; {
94    if(CCWRITE(fd, &c, 1)<1) return EOF;
95    else if(c==CR) {
96      if(CCWRITE(fd, "\12", 1)<1) return EOF;
97      }
98    return(c&255);
99    }
100
101 putchar(c)  char c;
102   (return (fputc(c, stdout));)
103
104 fputs(string, fd)  char string[]; int fd; {
105   int i, k; i=0; k=-1;
106   while(string[i]) {
107     while(string[++k] >= ' ');
108     if(string[k]==NULL) --k;
109     if(CCWRITE(fd, string+i, k-i+1) < k-i+1) return EOF;
110     if(string[k]=='\n') {
111       if(CCWRITE(fd, "\12", 1) < 1) return EOF;
112       }
113     i=k+1;
114     }
115   return NULL;
116   }
117
118 puts(string)  char string[];
119   (fputs(string, stdout); fputc(CR, stdout);)
120
121 fopen(name,mode)  char *name, *mode;
122   { int xfermode,fd;
123   if(mode==NULL) xfermode=0;
124     else if(*mode=='u') xfermode=3;
125     else if(*mode=='w')xfermode=2;
126     else xfermode=1;
127   fd=LST_PORT;
128   while(fd<LST_FILE) {
129     if(CCMODE(++fd)!=0) continue;
130     if(CCOPEN(fd, name, xfermode, 0, 0, 0)==0) return fd;
131     break;
132     }
133   return NULL;
```

```
48          DAD     SP
49          MOV     D,H
50          MOV     E,L
51          CALL    CCGCHAR
52          DCX     H
53          MOV     A,L
54          STAX    D
55          RET
56  ;
57  CCINCC: ENTRY
58          INX     H
59          INX     H
60          DAD     SP
61          MOV     D,H
62          MOV     E,L
63          CALL    CCGCHAR
64          INX     H
65          MOV     A,L
66          STAX    D
67          RET
68  ;
69  CCDDPDPC:ENTRY
70          DAD     D
71  CCPDPC: ENTRY
72          POP     B               ;;RET ADDR
73          POP     D
74          PUSH    B
75  ;
76  ;STORE A SINGLE BYTE FROM HL AT THE ADDRESS IN DE
77  CCPCHAR: ENTRY
78  PCHAR:  MOV     A,L
79          STAX    D
80          RET
81  ;
82  CCDECI: ENTRY
83          INX     H
84          INX     H
85          DAD     SP
86          MOV     D,H
87          MOV     E,L
88          CALL    CCGINT
89          DCX     H
90          JMP     CCPINT
91  ;
92  CCINCI: ENTRY
93          INX     H
94          INX     H
95          DAD     SP
96          MOV     D,H
97          MOV     E,L
98          CALL    CCGINT
99          INX     H
100         JMP     CCPINT
101 ;
102 CCDDPDPI:ENTRY
103         DAD     D
104 CCPDPI: ENTRY
105         POP     B               ;;RET ADDR
106         POP     D
107         PUSH    B
108 ;
109 ;STORE A 16-BIT INTEGER IN HL AT THE ADDRESS IN DE
110 CCPINT: ENTRY
```

```
134   )
135
136 fclose(fd)  int fd;
137   {
138   char c; c=255;
139   if((CCMODE(fd)==2)&(CCFILE(fd)!=0)) {
140     CCWRITE(fd, &c, 1);
141     }
142   CCCLOSE(fd);
143   }
144
145 getarg(n, str, maxsz, argc, argv) int n; char *str; int maxsz,argc, *argv;(
146   char *dptr, *sptr; int cnt;
147   if((n>(argc-1))|(n<0)) (str[0]=NULL; return EOF;)
148   cnt=0; dptr=str; sptr=argv[n];
149   while(*dptr++ = *sptr++) {
150     if(++cnt >= (maxsz-1))
151       (*dptr=NULL; break;)
152     }
153   return cnt;
154 )
```

Listing Three

```
 1
 2          #asm
 3 ;----- call.a: Small-C arithmetic and logical library
 4 ;
 5 CCDDGC:  ENTRY
 6          DAD     D
 7          JMP     CCGCHAR
 8 ;
 9 CCDSGC:  ENTRY
10          INX     H
11          INX     H
12          DAD     SP
13 ;
14 ;FETCH A SINGLE BYTE FROM THE ADDRESS IN HL AND
15 ; SIGN EXTEND INTO HL
16 CCGCHAR: ENTRY
17          MOV     A,M
18 ;PUT THE ACCUM INTO HL AND SIGN EXTEND THROUGH H.
19 CCARGC:  ENTRY
20 CCSXT:   ENTRY
21          MOV     L,A
22          RLC
23          SBB     A
24          MOV     H,A
25          RET
26 ;
27 CCDDGI:  ENTRY
28          DAD     D
29          JMP     CCGINT
30 ;
31 CCDSGI:  ENTRY
32          INX     H
33          INX     H
34          DAD     SP
35 ;
36 ;FETCH A FULL 16-BIT INTEGER FROM THE ADDRESS IN HL
37 ;INTO HL
38 CCGINT:  ENTRY
39          MOV     A,M
40          INX     H
41          MOV     H,M
42          MOV     L,A
43          RET
44 ;
45 CCDECC:  ENTRY
46          INX     H
47          INX     H
111 PINT:   MOV     A,L
112         STAX    D
113         INX     D
114         MOV     A,H
115         STAX    D
116         RET
117 ;INCLUSIVE "OR" HL AND DE INTO HL
118 CCOR:   ENTRY
119         MOV     A,L
120         ORA     E
121         MOV     L,A
122         MOV     A,H
123         ORA     D
124         MOV     H,A
125         RET
126 ;EXCLUSIVE "OR" HL AND DE INTO HL
127 CCXOR:  ENTRY
128         MOV     A,L
129         XRA     E
130         MOV     L,A
131         MOV     A,H
132         XRA     D
133         MOV     H,A
134         RET
135 ;"AND" HL AND DE INTO HL
136 CCAND:  ENTRY
137         MOV     A,L
138         ANA     E
139         MOV     L,A
140         MOV     A,H
141         ANA     D
142         MOV     H,A
143         RET
144 ;
145 ;IN ALL THE FOLLOWING COMPARE ROUTINES, HL IS SET TO 1 IF THE
146 ;CONDITION IS TRUE, OTHERWISE IT IS SET TO 0 (ZERO).
147 ;
148 ;TEST IF HL = DE
149 CCEQ:   ENTRY
150         CALL    CCCMP
151         RZ
152         DCX     H
153         RET
154 ;TEST IF DE != HL
155 CCNE:   ENTRY
156         CALL    CCCMP
157         RNZ
158         DCX     H
159         RET
160 ;TEST IF DE > HL (SIGNED)
161 CCGT:   ENTRY
162         XCHG
163         CALL    CCCMP
164         RC
165         DCX     H
166         RET
167 ;TEST IF DE <= HL (SIGNED)
168 CCLE:   ENTRY
169         CALL    CCCMP
170         RZ
171         RC
172         DCX     H
173         RET
174 ;TEST IF DE >= HL (SIGNED)
175 CCGE:   ENTRY
```

Small-C Compiler, v.2

Listing Three (Continued)

```
176             CALL    CCCMP
177             RNC
178             DCX     H
179             RET
180 ;TEST IF DE < HL (SIGNED)
181 CCLT:       ENTRY
182             CALL    CCCMP
183             RC
184             DCX     H
185             RET
186 ;COMMON ROUTINE TO PERFORM A SIGNED COMPARE
187 ; OF DE AND HL
188 ;THIS ROUTINE PERFORMS DE - HL AND SETS THE CONDITIONS:
189 ; CARRY REFLECTS SIGN OF DIFFERENCE (SET MEANS DE < HL)
190 ; ZERO/NON-ZERO SET ACCORDING TO EQUALITY.
191 CCCMP:      ENTRY
192             MOV     A,H             ;;INVERT SIGN OF HL
193             XRI     80H
194             MOV     H,A
195             MOV     A,D             ;;INVERT SIGN OF DE
196             XRI     80H
197             CMP     H               ;;COMPARE MSBS
198             JNZ     CCCMP1          ;;DONE IF NEQ
199             MOV     A,E             ;;COMPARE LSBS
200             CMP     L
201 CCCMP1:     LXI     H,1             ;;PRESET TRUE COND
202             RET
203 ;
204 ;TEST IF DE >= HL (UNSIGNED)
205 CCUGE:      ENTRY
206             CALL    CCUCMP
207             RNC
208             DCX     H
209             RET
210 ;
211 ;TEST IF DE < HL (UNSIGNED)
212 CCULT:      ENTRY
213             CALL    CCUCMP
214             RC
215             DCX     H
216             RET
217 ;
218 ;TEST IF DE > HL (UNSIGNED)
219 CCUGT:      ENTRY
220             XCHG
221             CALL    CCUCMP
222             RC
223             DCX     H
224             RET
225 ;
226 ;TEST IF DE <= HL (UNSIGNED)
227 CCULE:      ENTRY
228             CALL    CCUCMP
229             RZ
230             RC
231             DCX     H
232             RET
233 ;
234 ;COMMON ROUTINE TO PERFORM UNSIGNED COMPARE
235 ;CARRY SET IF DE < HL
236 ;ZERO/NONZERO SET ACCORDINGLY
237 CCUCMP:     ENTRY
```

```
307             ORA     B
308             RZ
309             XRA     A
310             MOV     A,E
311             RAL
312             MOV     E,A
313             MOV     A,D
314             RAL
315             MOV     D,A
316             ORA     E
317             RZ
318             JMP     CCMULTI
319 ;DIVIDE DE BY HL AND RETURN QUOTIENT IN HL, REMAINDER IN DE
320 ;(SIGNED DIVIDE)
321 CCDIV:      ENTRY
322 DIV:       MOV     B,H
323             MOV     C,L
324             MOV     A,D
325             XRA     B
326             PUSH    PSW
327             MOV     A,D
328             ORA     A
329             CM      CCDENEG
330             MOV     A,B
331             ORA     A
332             CM      CCBCNEG
333             MVI     A,16
334             PUSH    PSW
335             XCHG
336             LXI     D,0
337 CCDIV1:     DAD     H
338             CALL    CCRDEL
339             JZ      CCDIV2
340             CALL    CCCMPBCDE
341             JM      CCDIV2
342             MOV     A,L
343             ORI     1
344             MOV     L,A
345             MOV     A,E
346             SUB     C
347             MOV     E,A
348             MOV     A,D
349             SBB     B
350             MOV     D,A
351 CCDIV2:     POP     PSW
352             DCR     A
353             JZ      CCDIV3
354             PUSH    PSW
355             JMP     CCDIV1
356 CCDIV3:     POP     PSW
357             RP
358             CALL    CCDENEG
359             XCHG
360             CALL    CCDENEG
361             XCHG
362             RET
363 ;NEGATE THE INTEGER IN DE
364 ;(INTERNAL ROUTINE)
365 CCDENEG:    MOV     A,D
366             CMA
367             MOV     D,A
368             MOV     A,E
369             CMA
370             MOV     E,A
371             INX     D
372             RET
```

```
238             MOV     A,D
239             CMP     H
240             JNZ     CCUCMP1
241             MOV     A,E
242             CMP     L
243 CCUCMP1:    LXI     H,1
244             RET
245 ;
246 ;SHIFT DE ARITHMETICALLY RIGHT BY HL AND RETURN IN HL
247 CCASR:      ENTRY
248             XCHG
249             DCR     E
250             RM
251             MOV     A,H
252             RAL
253             MOV     A,H
254             RAR
255             MOV     H,A
256             MOV     A,L
257             RAR
258             MOV     L,A
259             JMP     CCASR+1
260 ;SHIFT DE ARITHMETICALLY LEFT BY HL AND RETURN IN HL
261 CCASL:      ENTRY
262             XCHG
263             DCR     E
264             RM
265             DAD     H
266             JMP     CCASL+1
267 ;SUBTRACT HL FROM DE AND RETURN IN HL
268 CCSUB:      ENTRY
269             MOV     A,E
270             SUB     L
271             MOV     L,A
272             MOV     A,D
273             SBB     H
274             MOV     H,A
275             RET
276 ;FORM THE TWO'S COMPLEMENT OF HL
277 CCNEG:      ENTRY
278             CALL    CCCOM
279             INX     H
280             RET
281 ;FORM THE ONE'S COMPLEMENT OF HL
282 CCCOM:      ENTRY
283             MOV     A,H
284             CMA
285             MOV     H,A
286             MOV     A,L
287             CMA
288             MOV     L,A
289             RET
290 ;MULTIPLY DE BY HL AND RETURN IN HL
291 ;(SIGNED MULTIPLY)
292 CCMULT:     ENTRY
293 MULT:       MOV     B,H
294             MOV     C,L
295             LXI     H,0
296 CCMULT1:    MOV     A,C
297             RRC
298             JNC     CCMULT2
299             DAD     D
300 CCMULT2:    XRA     A
301             MOV     A,B
302             RAR
303             MOV     B,A
304             MOV     A,C
305             RAR
306             MOV     C,A
```

(Continued on page 59, column 2)

```
373 ;NEGATE THE INTEGER IN BC
374 ;(INTERNAL ROUTINE)
375 CCBCNEG:    MOV     A,B
376             CMA
377             MOV     B,A
378             MOV     A,C
379             CMA
380             MOV     C,A
381             INX     B
382             RET
383 ;ROTATE DE LEFT ONE BIT
384 ;(INTERNAL ROUTINE)
385 CCRDEL:     MOV     A,E
386             RAL
387             MOV     E,A
388             MOV     A,D
389             RAL
390             MOV     D,A
391             ORA     E
392             RET
393 ;COMPARE BC TO DE
394 ;(INTERNAL ROUTINE)
395 CCCMPBCDE:  MOV     A,E
396             SUB     C
397             MOV     A,D
398             SBB     B
399             RET
400 ;
401 ; LOGICAL NEGATION
402 CCLNEG:     ENTRY
403             MOV     A,H
404             ORA     L
405             JNZ     $+6
406             MVI     L,1
407             RET
408             LXI     H,0
409             RET
410 ;
411 ; EXECUTE "SWITCH" STATEMENT
412 ;
413 ;  HL  =  SWITCH VALUE
414 ; (SP) -> SWITCH TABLE
415 ;           DW ADDR1, VALUE1
416 ;           DW ADDR2, VALUE2
417 ;           ...
418 ;           DW 0
419 ;          [JMP default]
420 ;           continuation
421 ;
422 CCSWITCH:  ENTRY
423             XCHG                    ;;DE =  SWITCH VALUE
424             POP     H               ;;HL -> SWITCH TABLE
425 SWLOOP:     MOV     C,M
426             INX     H
427             MOV     B,M             ;;BC -> CASE ADDR, ELSE 0
428             INX     H
429             MOV     A,B
430             ORA     C
431             JZ      SWEND           ;;DEFAULT OR CONTINUATION CODE
432             MOV     A,M
433             INX     H
434             CMP     E
435             MOV     A,M
436             INX     H
437             JNZ     SWLOOP
```

(Continued on next page)

Small-C Compiler, v.2

Listing Three (Continued)

```
438             CMP     D
439             JNZ     SWLOOP
440             MOV     H,B             ;;CASE MATCHED
441             MOV     L,C
442 SWEND:      PCHL
443 ;
444             #endasm
```

Listing Four

```
  1 /*
  2 ** stdio.h -- header for resident STDIO/CALL, and VM interfaces
  3 */
  4 #define stdin 0
  5 #define stdout 1
  6 #define stderr 2
  7 #define stdport 3
  8 #define stdlist 4
  9 #define CCEOM 45062
 10 #define ERR -2
 11 #define EOF -1
 12 #define YES 1
 13 #define NO 0
 14 #define NULL 0
 15 #define CR 13
 16 #define LF 10
 17 #asm
 18 STDIO:EQU 0E00H;;STDIO BASE
 19
 20 CCBOJ:  EQU STDIO+0006H
 21 CCEXIT: EQU STDIO+0009H
 22 CCPOLL: EQU STDIO+000CH
 23 CCHALT: EQU STDIO+000FH
 24 CCOPEN: EQU STDIO+0012H
 25 CCCLOSE:EQU STDIO+0015H
 26 CCREAD: EQU STDIO+0018H
 27 CCWRITE:EQU STDIO+001BH
 28 CCSEEK: EQU STDIO+001EH
 29 CCMODE: EQU STDIO+0021H
 30 CCFILE: EQU STDIO+0024H
 31 CCPURGE:EQU STDIO+0027H
 32 CCFLUSH:EQU STDIO+002AH
 33 CCALLOC:EQU STDIO+002DH
 34 CCFREE: EQU STDIO+0030H
 35 CCAVAIL:EQU STDIO+0033H
 36 CCDELIM:EQU STDIO+0036H
 37
 38 CCDDGC: EQU STDIO+0039H
 39 CCDSGC: EQU STDIO+003DH
 40 CCGCHAR:EQU STDIO+0040H
 41 CCARGC: EQU STDIO+0041H
 42 CCSXT:  EQU STDIO+0041H
 43 CCDDGI: EQU STDIO+0046H
 44 CCDSGI: EQU STDIO+004AH
 45 CCGINT: EQU STDIO+004DH
 46 CCDECC: EQU STDIO+0052H
 47 CCINCC: EQU STDIO+005EH
 48 CCDDPDPC:EQU STDIO+006AH
 49 CCPDPC: EQU STDIO+006BH
 50 CCPCHAR:EQU STDIO+006EH
 51 CCDECI: EQU STDIO+0071H
 52 CCINCI: EQU STDIO+007DH
 53 CCDDPDPI:EQU STDIO+0089H
```

```
 23 PUTC:   EXTRN
 24 PUTCHAR:EXTRN
 25 FPUTC:  EXTRN
 26 FPUTS:  EXTRN
 27 PUTS:   EXTRN
 28 FOPEN:  EXTRN
 29 FCLOSE: EXTRN
 30 GETARG: EXTRN
 31 ; STD I/O ROUTINES MATCHING VM ROUTINES
 32 FFLUSH: EXTRN
 33 ABORT:  EXTRN
 34 EXIT:   EXTRN
 35 UNLINK: EXTRN
 36 ; ARITHMETIC & LOGICAL ROUTINES
 37 CCARGC: EXTRN
 38 CCSXT:  EXTRN
 39 CCDSGI: EXTRN
 40 CCDDGI: EXTRN
 41 CCINCI: EXTRN
 42 CCDECI: EXTRN
 43 CCGINT: EXTRN
 44 CCDDPDPI:EXTRN
 45 CCPDPI: EXTRN
 46 CCDSGC: EXTRN
 47 CCDDGC: EXTRN
 48 CCINCC: EXTRN
 49 CCDECC: EXTRN
 50 CCGCHAR:EXTRN
 51 CCDDPDPC:EXTRN
 52 CCPDPC: EXTRN
 53 CCPCHAR:EXTRN
 54 CCPINT: EXTRN
 55 CCOR:   EXTRN
 56 CCXOR:  EXTRN
 57 CCAND:  EXTRN
 58 CCEQ:   EXTRN
 59 CCNE:   EXTRN
 60 CCGT:   EXTRN
 61 CCLE:   EXTRN
 62 CCGE:   EXTRN
 63 CCLT:   EXTRN
 64 CCCMP:  EXTRN
 65 CCUGE:  EXTRN
 66 CCULT:  EXTRN
 67 CCUGT:  EXTRN
 68 CCULE:  EXTRN
 69 CCUCMP: EXTRN
 70 CCASR:  EXTRN
 71 CCASL:  EXTRN
 72 CCSUB:  EXTRN
 73 CCNEG:  EXTRN
 74 CCCOM:  EXTRN
 75 CCMULT: EXTRN
 76 CCDIV:  EXTRN
 77 CCLNEG: EXTRN
 78 CCSWITCH:EXTRN
 79 ; VM ROUTINES
 80 VM.R:   LIBRY
 81 CCBOJ:  EXTRN
 82 CCEXIT: EXTRN
 83 CCPOLL: EXTRN
 84 CCHALT: EXTRN
 85 CCOPEN: EXTRN
 86 CCCLOSE:EXTRN
 87 CCDELIM:EXTRN
 88 CCREAD: EXTRN
 89 CCWRITE:EXTRN
 90 CCSEEK: EXTRN
```

```
54 CCPDPI: EQU STDIO+008AH
55 CCPINT: EQU STDIO+008DH
56 CCOR:   EQU STDIO+0093H
57 CCXOR:  EQU STDIO+009AH
58 CCAND:  EQU STDIO+00A1H
59 CCEQ:   EQU STDIO+00A8H
60 CCNE:   EQU STDIO+00AEH
61 CCGT:   EQU STDIO+00B4H
62 CCLE:   EQU STDIO+00BBH
63 CCGE:   EQU STDIO+00C2H
64 CCLT:   EQU STDIO+00C8H
65 CCCMP:  EQU STDIO+00CEH
66 CCUGE:  EQU STDIO+00DFH
67 CCULT:  EQU STDIO+00E5H
68 CCUGT:  EQU STDIO+00EBH
69 CCULE:  EQU STDIO+00F2H
70 CCUCMP: EQU STDIO+00F9H
71 CCASR:  EQU STDIO+0104H
72 CCASL:  EQU STDIO+0112H
73 CCSUB:  EQU STDIO+0119H
74 CCNEG:  EQU STDIO+0120H
75 CCCOM:  EQU STDIO+0125H
76 CCMULT: EQU STDIO+012CH
77 CCDIV:  EQU STDIO+014CH
78 CCLNEG: EQU STDIO+01AAH
79 CCSWITCH:EQU STDIO+01B6H
80
81 GETC:   EQU STDIO+01D0H
82 PUTC:   EQU STDIO+01D3H
83 FFLUSH: EQU STDIO+01D6H
84 ABORT:  EQU STDIO+01D9H
85 EXIT:   EQU STDIO+01DCH
86 UNLINK: EQU STDIO+01DFH
87 FGETC:  EQU STDIO+01E2H
88 GETCHAR:EQU STDIO+02D9H
89 FGETS:  EQU STDIO+02E2H
90 FPUTC:  EQU STDIO+050DH
91 PUTCHAR:EQU STDIO+0579H
92 FPUTS:  EQU STDIO+058AH
93 PUTS:   EQU STDIO+0699H
94 FOPEN:  EQU STDIO+06B5H
95 FCLOSE: EQU STDIO+078BH
96 GETARG: EQU STDIO+07E7H
97 #endasm
```

Listing Five

```
 1 /*
 2 ** stdiol.h -- header for local STDIO/CALL, and VM interfaces
 3 */
 4 #define stdin 0
 5 #define stdout 1
 6 #define stderr 2
 7 #define stdport 3
 8 #define stdlist 4
 9 #define ERR -2
10 #define EOF -1
11 #define YES 1
12 #define NO 0
13 #define NULL 0
14 #define CR 13
15 #define LF 10
16 #asm
17 STDIOL.R:LIBRY
18 ; STD I/O ROUTINES
19 GETC:   EXTRN
20 GETCHAR:EXTRN
21 FGETC:  EXTRN
22 FGETS:  EXTRN
```

(Continued on page 61, column 2)

```
91 CCMODE: EXTRN
92 CCFILE: EXTRN
93 CCPURGE:EXTRN
94 CCFLUSH:EXTRN
95 CCALLOC:EXTRN
96 CCFREE: EXTRN
97 CCAVAIL:EXTRN
98 #endasm
```

Listing Six

```
 1
 2 /*
 3 ** lib.c -- function library
 4 **
 5 ** Copyright 1982 J. E. Hendrix
 6 */
 7 #define NOCCARGC /* don't pass arg count to functions */
 8                  /* called by these functions */
 9 #include abs.c,2
 1
 2 /*
 3 ** abs -- returns absolute value of nbr
 4 */
 5 abs(nbr)  int nbr;
 6   {if(nbr<0) return -nbr; else return nbr;}
10 #include dtoi.c,2
20     }
21   if(sz) str[--sz]=sgn;
22   while(sz>0) str[--sz]=' ';
23   return str;
24   }
12 #include itou.c,2
 1
 2 /*
 3 ** itou -- convert nbr to unsigned decimal string of width sz
 4 **         right adjusted, blank filled; returns str
 5 **
 6 **        if sz > 0 terminate with null byte
 7 **        if sz = 0 find end of string
 8 **        if sz < 0 use last byte for data
 9 */
10 itou(nbr, str, sz)  int nbr;  char str[];  int sz;  {
11   int lowbit;
12   if(sz>0) str[--sz]=NULL;
13   else if(sz<0) sz = -sz;
14   else while(str[sz]!=NULL) ++sz;
15   while(sz) {
16     lowbit=nbr&1;
17     nbr=(nbr>>1)&32767;  /* divide by 2 */
18     str[--sz]=((nbr%5)<<1)+lowbit+'0';
19     if((nbr=nbr/5)==0) break;
20     }
21   while(sz) str[--sz]=' ';
22   return str;
23   }
13 #include itox.c,2
 1
 2 /*
 3 ** itox -- converts nbr to hex string of length sz
 4 **         right adjusted and blank filled, returns str
 5 **
 6 **        if sz > 0 terminate with null byte
 7 **        if sz = 0 find end of string
 8 **        if sz < 0 use last byte for data
 9 */
10 itox(nbr, str, sz)  int nbr;  char str[];  int sz;  {
```

(Continued on next page)

Small-C Compiler, v.2

Listing Six (Continued)

```
11    int digit, offset;
12    if(sz>0) str[--sz]=NULL;
13    else if(sz<0) sz = -sz;
14    else while(str[sz]!=NULL) ++sz;
15    while(sz) {
16      digit=nbr&15; nbr=(nbr>>4)&4095;
17      if(digit<10) offset=48; else offset=55;
18      str[--sz]=digit+offset;
19      if(nbr==0) break;
20      }
21    while(sz) str[--sz]=' ';
22    return str;
23    }
14  #include left.c,2
 1
 2  /*
 3  ** left -- left adjust and null terminate a string
 4  */
 5  left(str) char *str; {
 6    char *str2;
 7    str2=str;
 8    while(*str2==' ') ++str2;
 9    while(*str++ = *str2++);
10    }
15  #include printf.c,2
 1
 2  /*
 3  ** printf(controlstring, arg, arg, ...) -- formatted print
 4  **         operates as described by Kernighan & Ritchie
 5  **         only d, x, c, s, and u specs are supported.
 6  */
 7  printf(argc) int argc; {
 8    int i, width, prec, preclen, len, *nxtarg;
 9    char *ctl, *cx, c, right, str[7], *sptr, pad;
10    i = CCARGC();    /* fetch arg count from A reg first */
11    nxtarg = &argc + i - 1;
12    ctl = *nxtarg;
13    while(c=*ctl++) {
14      if(c!='%') {cout(c, stdout); continue;}
15      if(*ctl=='%') {cout(*ctl++, stdout); continue;}
16      cx=ctl;
17      if(*cx=='-') {right=0; ++cx;} else right=1;
18      if(*cx=='0') {pad='0'; ++cx;} else pad=' ';
19      if((i=utoi(cx, &width)) >= 0) cx=cx+i; else continue;
20      if(*cx=='.') {
21        if((preclen=utoi(++cx, &prec)) >= 0) cx=cx+preclen;
22        else continue;
23        }
24      else preclen=0;
25      sptr=str; c=*cx++; i=*(--nxtarg);
26      if(c=='d') itod(i, str, 7);
27      else if(c=='x') itox(i, str, 7);
28      else if(c=='c') {str[0]=i; str[1]=NULL;}
29      else if(c=='s') sptr=i;
30      else if(c=='u') itou(i, str, 7);
31      else continue;
32      ctl=cx; /* accept conversion spec */
33      if(c!='s') while(*sptr==' ') ++sptr;
34      len=-1; while(sptr[++len]); /* get length */
35      if((c=='s')&(len>prec)&(preclen>0)) len=prec;
36      if(right) while(((width--)-len)>0) cout(pad, stdout);
37      while(len) {cout(*sptr++, stdout); --len; --width;}
38      while(((width--)-len)>0) cout(pad, stdout);
12      d++; *nbr=t;
13      }
14    return d;
15    }
19  #include xtoi.c,2
 1
 2  /*
 3  ** xtoi -- convert hex string to integer nbr
 4  **         returns field size, else ERR on error
 5  */
 6  xtoi(hexstr, nbr)  char *hexstr;  int *nbr;  {
 7    int d,t; d=0;
 8    *nbr=0;
 9    while(1)
10      {
11      if((*hexstr>='0')&(*hexstr<='9')) t=48;
12      else if((*hexstr>='A')&(*hexstr<='F')) t=55;
13      else if((*hexstr>='a')&(*hexstr<='f')) t=87;
14      else break;
15      if(d<4) ++d; else return ERR;
16      *nbr=*nbr<<4;
17      *nbr=*nbr+(*hexstr++)-t;
18      }
19    return d;
20    }
20  #include out.c,2
 1
 2  /*
 3  ** out.c -- output to fd checking for errors
 4  */
 5  cout(c, fd) char c; int fd; {
 6    if(fputc(c, fd)==EOF) xout();
 7    }
 8
 9  sout(string, fd) char *string; int fd; {
10    if(fputs(string, fd)==EOF) xout();
11    }
12
13  lout(line, fd) char *line; int fd; {
14    sout(line, fd);
15    cout('\n', fd);
16    }
17
18  xout() {
19    fputs("output error\n", stderr);
20    abort(7);
21    }
```

End Listing

```
39      )
40    )

16 #include sign.c,2
 1
 2 /*
 3 ** sign -- return -1, 0, +1 depending on the sign of nbr
 4 */
 5 sign(nbr)  int nbr;  (
 6   if(nbr>0) return 1;
 7   else if(nbr==0) return 0;
 8   else return -1;
 9   )

17 #include strcmp.c,2
 1
 2 /*
 3 ** strcmp -- return -1, 0, +1 depending on str1 <, =, > str2
 4 */
 5 strcmp(str1, str2)  char *str1, *str2;  (
 6   char c1, c2;
 7   while((c1=*str1++)==(c2=*str2++)) if(c1==NULL) return 0;
 8   return sign(c1-c2);
 9   )
18 #include utoi.c,2
 1
 2 /*
 3 ** dtoi -- convert signed decimal string to integer nbr
 4 **         returns field length, else ERR on error
 5 */
 6 dtoi(decstr, nbr)  char *decstr;  int *nbr;  (
 7   int len, s;
 8   if((*decstr)=='-') (s=1; ++decstr;) else s=0;
 9   if((len=utoi(decstr, nbr))<0) return ERR;
10   if(*nbr<0) return ERR;
11   if(s) (*nbr = -*nbr; return ++len;) else return len;
12   )
11 #include itod.c,2
 1
 2 /*
 3 ** itod -- convert nbr to signed decimal string of width sz
 4 **         right adjusted, blank filled; returns str
 5 **
 6 **        if sz > 0 terminate with null byte
 7 **        if sz = 0 find end of string
 8 **        if sz < 0 use last byte for data
 9 */
10 itod(nbr, str, sz)  int nbr;  char str[];  int sz;  (
11   char sgn;
12   if(nbr<0) (nbr = -nbr; sgn='-';)
13   else sgn=' ';
14   if(sz>0) str[--sz]=NULL;
15   else if(sz<0) sz = -sz;
16   else while(str[sz]!=NULL) ++sz;
17   while(sz) {
18     str[--sz]=(nbr%10+'0');
19     if((nbr=nbr/10)==0) break;

 1
 2 /*
 3 ** utoi -- convert unsigned decimal string to integer nbr
 4 **          returns field size, else ERR on error
 5 */
 6 utoi(decstr, nbr)  char *decstr;  int *nbr;  (
 7   int d,t; d=0;
 8   *nbr=0;
 9   while((*decstr>='0')&(*decstr<='9')) {
10     t=*nbr;t=(10*t) + (*decstr++ - '0');
11     if ((t>=0)&(*nbr<0)) return ERR;
```

(Continued on page 63, column 2)

CP/M EXCHANGE

by Gene Head

I can now supply many standard modem programs on disk. All source programs are configured for serial modem connected to an IMSAI MIO (Multiple I/O). *[Editor's note: We still haven't found anything better at twice the price. These MIO's, however, no longer seem to be available from Fischer-Frietas-IMSAI in Oakland, California.]*

I also have a hot DIR.ASM program available which will replace simple DIR in the CCP. You can name it DIR.COM or DIR.CMD. This program is quite similar to the LST program in the new CP/M86. It displays the directory after an Alpha sort and displays information in the footer much as if you had used STAT *.*. The footer line looks like this:

Drive: Xn Files (Number requested)
Bytes: nK (Used for each file)

With a little tooling around, one could get the actual keystroke sequence down to one or two characters and a carriage return.

The entire listing will appear as an article in *DDJ* soon but in the meantime if you want a listing or any of the Modem programs, send the stamps sufficient to cover postage, your disk and mailer and a cover letter along with five bucks, and I will ship the material off right away first class. My address is:

Gene Head
Headquarters
2860 N.W. Skyline Drive
Corvallis, Oregon 97330

Or call my RCP/M system:

RCP/M Exchange System
Ringback 9:00 a.m. – 9:00 p.m.
Monday – Friday only
(503) 758-8408
300 baud only

Paul Pennington (2912 Palmetto Drive, Martinez, GA 30907) says he has won the battle of MODEM7 vs. the Heath 89/90 computer. Send him a disk, a mailer, and five dollars to cover postage and his effort, and he will return the disk with MODEM7 configured for the Heath and several documentation files that should make modem communications easier. Paul can handle any Heath five-inch format as well as SSSD IBM eight-inch disks (CP/M *not* HDOS).

H. G. Heard (50 Skywood Way, Woodside, CA 94062) wants help interfacing a TRS-80 (port B) with an Anderson/Jacobson ADAC 242 modem and the MODEM7 software package.

John Beal (15 Dumont Round, Belle Mead, NJ 08502) is looking for any experienced CP/M users that he can meet in his area for software and hardware assistance.

DDJ

OF INTEREST

by Michael Wiesenberg

Nuts and Bolts

"**MICROMINT MPX-16** takes on the IBM PC," says the press release announcing the $1200 (in quantities of 100) single-board computer system fully compatible with hardware peripherals and software for the IBM PC. It has a 5MHz 8088 with 20-bit addressing that allows up to 1Mb of memory. An 8087 math coprocessor and hard disk drive can be added. The board comes with 256K RAM, two serial and three parallel ports, double-density floppy disk controller, and five expansion slots. It uses CP/M-86, but MS-DOS and "other software" are coming "soon." The CP/M-86 BIOS is in EPROMs on the board. All you add are power supply, serial terminal, and one floppy drive. **Reader Service No. 205.**

For high-resolution color graphics processors, monitor, and printer, I'm not sure what "low-cost" and "affordable" (according to the release) mean, but the prices certainly are coming down. Vectrix Corporation's 8088-based **VX128** has 672-by-480 resolution, high-speed three-dimensional image generation, graphics commands, 128K graphics RAM, three-bit planes, and eight colors for $1995 (less than half that of the competition, they claim), while the **VX384** has 384K graphics RAM, nine-bit planes, and displays 512 simultaneous colors from a palette of 16 million for $3995. Both translate commands from most micros or minis (at 300 to 19.2K baud from serial or 8-bit parallel port), have a NEC PD7220/GDC chip that generates high-speed lines and arcs in many dot and dash patterns with 1600-nanosecond pixel update, video output in separate or combined horizontal and vertical or sync on green, 40 rows of 96 alphanumeric characters in a 5-by-9 matrix with descenders, software for point, line, arc, and polygon primitives in two and three dimensions, solid polygons with pattern fill, various 3-D commands, character generation with slant, magnification, and spacing parameters, and bidirectional DMA access to graphics RAM. The 13-inch high-resolution RGB **VXM** monitor costs $1295, as does the **IDS Prism** color printer. The printer can be accessed without affecting the display. Both graphics processors can be commanded from high-level and assembly languages. Interfaces are included for light pen and keyboard add-ons. **Reader Service No. 207.**

The **Compu/time Universal Floppy Disk Controller I** board from GSR Computers connects up to four drives in any combination of 5¼- or 8-inch ANSI standard interfaces or both. It reads and writes disks of different sizes and formats, from 5¼-inch, 40-track, single-sided, single-density to 96-track, double-sided, double-density with 512 bytes per sector, and 8-inch, single-sided, single-density to double-sided, double-density with 1024 bytes per sector. Since the source listing of the format program is included, you can also create your own formats. In addition to the board, you get a manual and 8-inch, single-sided, single-density CP/M disk with Monitor/BIOS and disk formatter source listings. The UFDC-1, assembled and tested, costs $325. A kit is $295, and bare board is $60. Add $5 shipping and handling, and residents of New York State add 8¼% sales tax. **Reader Service No. 209.**

Hewlett-Packard now has two new personal computers being sold through retail computer dealers. The **HP Series 100 Model 120** has the power of the HP 125 in a smaller package. It is a CP/M-based, 8-bit, dual-processor computer with 64K main memory, 16K screen display memory, and 32K fixed memory for internal functions. It has screen-labelled softkeys. Available software includes WordStar, SpellStar, MailMerge, VisiCalc, CONDOR DBMS and report generators, Series 100 WORD word processing (HP's implementation of Lexisoft's Spellbinder), Graphics, DSN/Link, and BASIC. The $2275 price includes processor, display, 64K memory, and keyboard. The **HP Series 200 Model 16** is HP's first 16-bit personal computer to be sold through dealer channels. Its processor is the Motorola 8MHz 16/32-bit 68000. Built-in graphics are standard, as are five user-definable softkeys (a number doubled with the shift key), an 80-column-by-25-row, 300-by-400-pixel display, and a rotary control knob for fast editing, cursor movement, analog instrument control, and other linear input applications. Three language systems are available: BASIC, HPL, and Pascal. Third-party vendors offer others, including FORTH. UNIX should soon be available, and CP/M is also planned. The Model 16 can be linked with other HP desktop computers in a network that shares disk drives and printers. Base price is $3650, for processor, display, 128K memory (expandable to 768K), and keyboard. Both models have a tilt-and-swivel accessory for $110. Available peripherals for both systems include dual 3½-inch micro-floppies with 270K per disk for $1775 (or one drive for $1200) and one micro-floppy drive plus 4.6Mb Winchester for $4975. HP also offers a daisy-wheel printer, the **HP 2602A**, for $1950. **Reader Service No. 211.**

Memory Merchant's 64K static RAM S-100 memory board operates at 6MHz or greater, has four independently addressable 16K blocks, uses 150ns 16K (2K x 8) CMOS static RAMS, has a two-year warranty and costs $629. **Reader Service No. 213.**

Add intelligent text and graphics printing capabilities to your Apple II or /// with **PKASO** (is that a pun?) from Interactive Structures Inc. Dump any screen, print in 26 shades of gray, use your own or predefined characters, and draw with both high- and low-resolution graphics on many dot matrix printers. Interface, software, cable, manual, and demo disk cost $165. **Reader Service No. 215.**

Do you have five computers and one printer? How about five printers and one computer? Handle either, and other permutations, with **MultiSpool** from Digital Laboratories Inc., a buffered multiport spooler with a Z80 processor, 64K DRAM, 8K ROM, 4 UART channels, various configurations of parallel 8-bit ports and serial ports, and status LEDs, priced from $595 to $995, depending on configuration. **Reader Service No. 217.**

Bits and Bytes

Want to do word processing on your VIC-20 or Commodore-64? **Quick Brown Fox** can be used even on the VIC-20's 22-column display. For $65 you get line and global editing, text manipulation, boilerplating, tab and margin settings, right justification, proportionally spaced printing, and automatic reformatting of edited text. **Reader Service No. 219.**

Elektrokonsult offers disk utilities for the Osborne-1 that avoid data loss. **DTEST** tests for bad sectors, and collects them in a write-protected file. **UNERA** recovers erased files. **DDUP** automatically recovers files with damaged sectors. **DDUMP** examines and patches data on any sector, enabling you to repair crashed disks and recover lost data. Each program costs $29.95, or you can get all four for $99.95. Add $8 for air shipment from Norway. **Reader Service No. 221.**

Want to use your IBM PC to talk to mainframes? **Blue Lynx** from Techland Systems Inc., a 3270 emulation package, currently emulates an IBM 3276 controller/display under SDLC protocol and will soon offer bisync protocol and IBM 3278 emulation for connection by coaxial cable to 3274-like cluster controllers. What you get is a plug-in communications card and software on floppy disk for $690. **Reader Service No. 231.**

Another telecommunications package from Century Systems, called **ComCrypt**, combines computer communications with encryption and decryption to send and receive CP/M files with complete security, using a proprietary encryption/decryption technique. ComCrypt provides users of the CP/M operating system with security features, including an encrypted "chat" mode, and helps to prevent unauthorized "eavesdropping" on computer communications. It operates over regular telephone lines or direct wire between computers. The installation program supports a large number of common terminals. To use ComCrypt on your machine requires a Z80, CP/M version 2.0 or later, a 24 x 80 terminal with clear screen and addressable cursor, and $250. **Reader Service No. 233.**

The **mbp COBOL** (ANSI 74) compiler from mbp Software, for several 8086-based systems, including CP/M-86, MS-DOS, and iRMX-86, with MP/M-86 and OASIS-16 coming soon, generates native code. It automatically corrects simple syntax errors. Included in the $500 price is a 45K run-time library. **Reader Service No. 235.**

The **ZAS Z-8000 Software Development Package** from Western Wares runs on any CP/M-80 system. The relocatable macro assembler supports both segmented and nonsegmented code, "include" files, nested conditional assembly, eight macro directives, and external CALR references. The ZLK Task Builder combines and renames program and data sections, accepts commands from console or command file, builds overlay programs, and can convert any or all program sections into absolute form. The ZLD object code manipulation utilities download, translate to Intel hex format, and memory load into dual-processor configurations. The ZEX run-time monitor, supplied in source and object form, supports any dual processor system that has CP/M-80. $395 gets you an 8-inch CP/M-80 formatted, single-density floppy that also runs on Cromemco's CDOS. **Reader Service No. 237.**

You can get Nevada COBOL, FORTRAN, PILOT, or EDIT from Ellis Computing for $29.95 each, "the same professional packages," they tell us, "that have sold for as much as $300 per copy." **Reader Service No. 239.**

Other Stuff

Could you use Radio Shack's help in selling your software to the TRS-80 computer market? The **Outside Software Support Program** is available to professional software developers, even those designing programs not intended for ultimate sale through Radio Shack outlets. The Technical Assistance Program provides information, licenses TRSDOS and TRS-80 BASIC, and sells TRSDOS source code and bulk run-time packages. The Software Review Program evaluates third-party software and makes available a listing of reviewed software at Radio Shack Computer Centers. Submitted software can also be considered for Radio Shack product lines. **Reader Service No. 241.**

You all want to "plan, build, debug, and program a working computer system!", don't you? (That's not *my* exclamation point!) You need Tab Books' *The Microcomputer Builder's Bible* by Chris Johnston. The book discusses all popular 8- and 16-bit chips, focusing on the 8080/Z80 processors, describes the S-100/IEEE-696 bus, and provides much software information, such as monitors and operating systems (CP/M in particular), languages, and graphics interfaces. Detailed illustrations and schematics describe building a computer on both board and system levels, along with advice on soldering, wire wrapping, constructing PC boards, and debugging. This 320-page book with 275 illustrations costs $18.95 in hardback and $12.95 in paper.

Tab Books also offers *From BASIC to Pascal* by Ronald Anderson, buiding on your knowledge of BASIC to learn Pascal. 322 pages, 38 illustrations, and $17.95/$10.95. **Reader Service No. 243.**

You'll also need complete information about all currently available microprocessors, so get the *Microprocessor Data Book* by S. A. Money from McGraw-Hill. All 4-, 8-, and 16-bit microprocessors from American, European, and Japanese manufacturers are covered, with illustrations and pinouts, descriptions of architecture, operating parameters, data about development aids and software, specifications, related chips by the same manufacturer, and a cross-reference of equivalent microprocessors. This 264-page, 8½-by-11 book costs $38. **Reader Service No. 245.**

Computer leasing seems to be a burgeoning trend in the computer business. **North Star Computers** now offers, through U.S. Leasing, three- to five-year leases on the Advantage, with system upgrades permitted. **Reader Service No. 247.**

Another trend appeared in the press notice from **Information Unlimited Software** announcing that Sears has joined their product distribution network. IUS features "professional" software, such as EasyWriter II and the rest of the Easy-Family and various financial packages. In addition to Sears Business Systems Centers, you can get IUS products in ComputerLand stores and all authorized IBM PC dealers. **Reader Service No. 249.**

Another recent trend is the requirement at various universities for all new students to provide their own computers. West Philadelphia's Drexel University was the first to announce that next fall's freshman class must all buy computers for classroom work. Drexel will probably require a computer costing about $1000. Carnegie-Mellon University of Pittsburgh, renowned for computer and robotics studies, recently announced that all entering freshmen in 1985 would have to supply their own computers. Clarkson College, of Potsdam, NY, will provide personal computers for all freshmen in fall, 1983. They have arranged to purchase over 1000 dual-processor Z100s a year from Zenith Data Systems for the next four years. Students will pay a one-time $200 maintenance deposit plus $200 per semester for use of their computers. Upon graduation, students will own their computers, although those who don't want them will get back the $200 deposit. The computers will be linked in a campus-wide network, and have CP/M-85, ZDOS, Z-BASIC with graphics, Fortran, COBOL, and Multiplan.

I probably won't be the first to tell you that the 8th West Coast Computer Faire will take place in the San Francisco Civic Auditorium and Brooks Hall complex March 18 to 20, 1983, but I can turn you on to a free subscription to the *Silicon Gulch Gazette*. **Reader Service No. 251.**

TUG Plug

We at *Dr. Dobb's* are pleased to an-

nounce the arrival of a new kid on the block, a magazine called *The User's Guide*, or *TUG* for short. This periodical has just been launched. The premiere issue looks great. *The User's Guide* is managed and edited by Tony Bove, who wrote the Sybex CP/M book, now known worldwide, and Cheryl Rhodes, a computer literacy instructor and programmer. *TUG* emphasizes classical and innovative art covers and tongue-in-cheek humor in the process of presenting tutorial material of interest to a wide spectrum of readers. *TUG* is not dedicated exclusively to tutorials but Tony and Cheryl and their staff see tutorials as an area of great importance especially since so many companies release their products with documentation lacking in true human factors engineering. A subscription can be obtained by enquiry to: *The User's Guide*, 117 Ware Road, Woodside, CA 94062.

FOG Bound

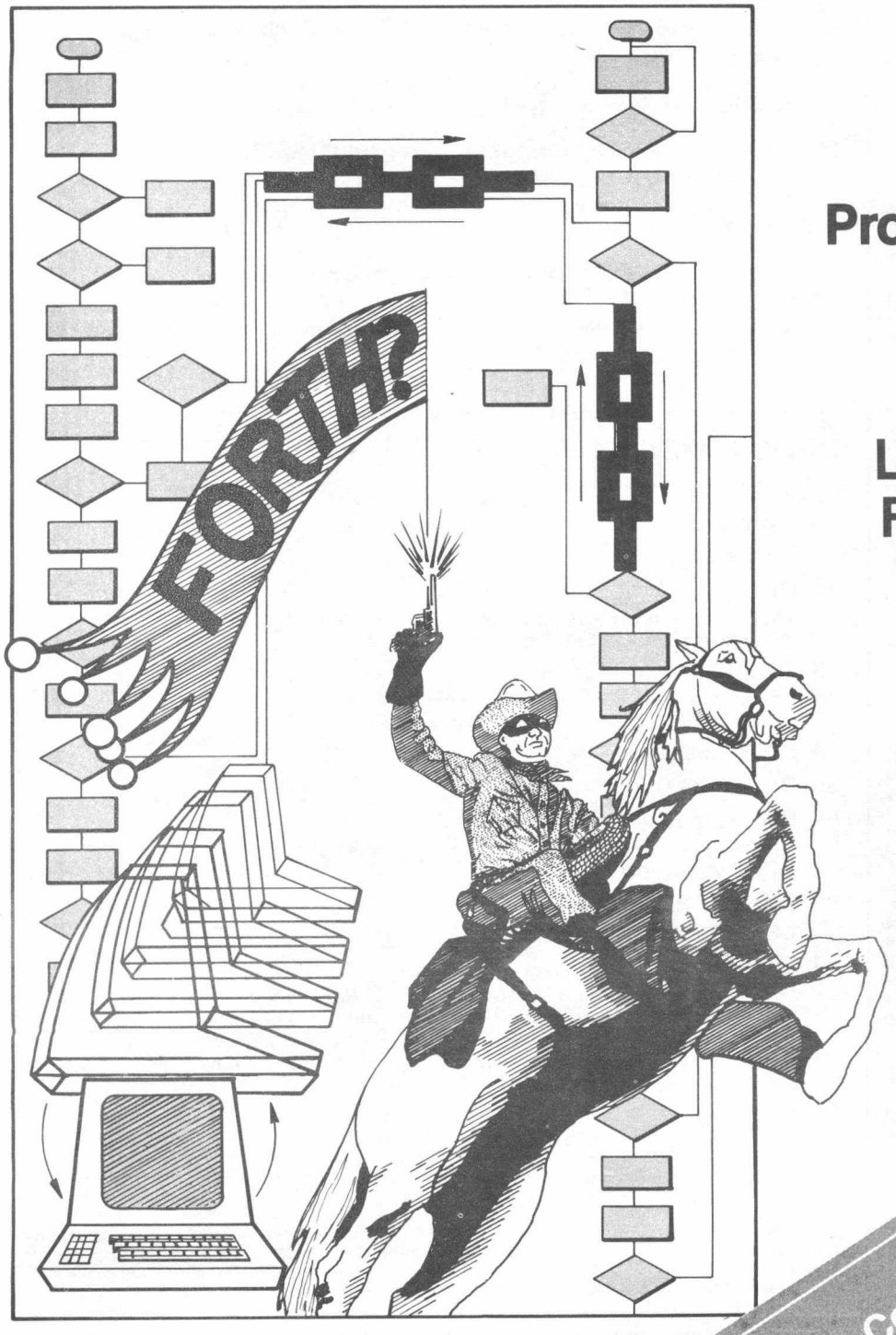
F.O.G., The First Osborne Group, formerly F.O.U.G., the First Osborne Users Group, meets on the third Thursday of every month at the Dysan Auditorium, Patrick Henry Drive, Santa Clara, California. If you use an Osborne, you will be pleased to make the acquaintance of these nice people. The club name was modified due to frequent mispronunciation of the acronym by various participants especially in heated discourse.

DDJ

#76 February 1983 $2.50 ($3.00 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems



PISTOL

Program Linkage by Coroutines

Linking CP/M Functions to High-Level Programs

Plus:
Concurrent CP/M-86
(See page 18)

For Users of Small Computer Systems

February 1983 Volume 8, Issue 2

Publisher/Art Director – Clifford West
Senior Editor – Hank Harrison
Associate Editor – Reynold Wiggins
Contributing Editors –
Dave Caulkins, Dave Cortesi,
Ray Duncan, Gene Head,
Michael Wiesenberg
Marketing Director – Craig Harper
Marketing Manager – Beatrice Blatteis
Advertising Director – Carl Landau
Ad Sales Representative – Doug Millison
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Terri Marty, Linda Marohn
Production Manager – Barbara Ruzgerian
Production Assistant – Jane A. McKean
Typesetter – Paula Fairchild
Cover Illustration – Al McCahon

© 1982 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus, Paul Lutus, Ben Goldfarb, Rodney Black, Steven Weiss, Pat Phelan, John Brodie, James Fleming, Unison World, Thomas Davis, Jan Steinman, Ronald E. Johnson, G. Hunter, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III. **Sustaining Subscribers:** Pete Roberts, Steven Fisher Enterprizes. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

CONTENTS

ARTICLES

12 PISTOL
A Forth-like Portably Implemented STack Oriented Language
By Ernest E. Bergmann
Dr. Bergmann, wishing to have Forth-like languages on a variety of machines, has designed this one with an eye to high portability, simplicity, and user friendliness.

26 Program Linkage by Coroutines – Forth to BASIC
By Leonard Morgenstern, M.D.
Using the general principles of coroutines, Dr. Morgenstern describes how he linked BASIC file-handling capabilities to a Forth graphics program.

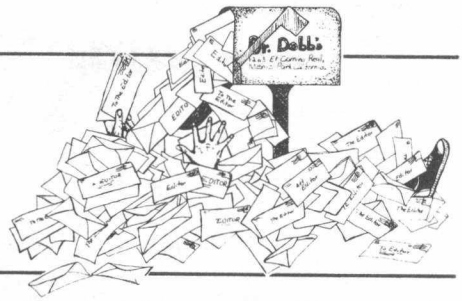
32 Linking CP/M Functions to Your High-Level Program
By Wendy A. Weiger
Author Weiger describes how to call CP/M Functions such as DEL from within a high-level program.

DEPARTMENTS

10 Letters
10 Editorial
16 Dobb's Ex Machina
The column returns with a potpourri of rumors and a bit of conjecture about the industry.
18 16-Bit Software Toolbox
An extensive review of the Concurrent CP/M-86 multi-tasking package from Digital Research.
34 CP/M Exchange
55 Dr. Dobb's Clinic
58 Hardware Review
Small Systems Engineering CP/M-80 Expansion Card for the Victor 9000.
62 Software Review
REVAS Disassembler
64 Book Reviews
Data Communications for Micro Computers; *TRS-80 Data Communications Systems*; *The Network Revolution: Confession of a Computer Scientist*; *The Online Micro-Software Guide and Directory 1983/84*; and *CRUNCHERS-21 – Simple Games for the Timex/Sinclair 1000 2x*.
68 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

LETTERS



High Praise for Small C

To the Readers of Dr. Dobb's:

The December issue of *Dr. Dobb's* containing the first installment of version 2 of the Small-C compiler was definitely a welcome sight. With all the bug corrections and improvements for version 1.0 which have been dribbling in these last couple years, it was becoming clear that someone would just have to sit down and pull together all the pieces.

Marlin Ouverson approached me with the idea of an official rewrite of the little compiler. Though I was eager to give it a shot, there was just no way to find all the spare time required to do it right. At the same time, it was clear (at least to me) that there were some pretty respectable Small-C experts emerging from the readership of *Dr. Dobb's* who were quite possibly more familiar with the original compiler than I (odd thought, that).

So Marlin and I pondered the problem and soon realized there was one name which had appeared more than once in regards to Small-C . . . J.E. Hendrix. But could we ask him to do it? Marlin contacted him and, well, you can see the results.

I want to make it quite clear that my only role in this was sending to Jim a rather open-ended wish list. From my suggestions (most of which he had already anticipated or even implemented), and from those of Neal Block and Dr. James Van Zandt, Jim somehow found the time necessary to fix the old bugs, check off our improvement wish lists, and toss in some compile-time options of his own. I consider the finished product an excellent product.

Some readers may wish for features not yet implemented (naturally, any programmer wants all the language features possible). Jim is as aware of those as anyone. However, my own feeling is that he chose exactly the right place to stop development, chase out the bugs, and get the thing into print. It is the same decision I made many months (and features) ago.

I have no doubt that this new version represents far more work than the original (which was less than a month of spare time in the cooking). And because of that, it is a much more generous gift.

Sincerely,
Ron Cain
200 Meadow Drive
Boulder Creek, CA 95006

Dear *DDJ*,

Congratulations! A first class effort! The December issue was worth my whole year's subscription. I was very impressed with Mr. Hendrix's method for evaluating constant expressions, and for dumping redundant code. His evaluation of the "switch/case/default" is much more elegant than the method I had devised for my own extensions. The whole article is an object lesson in the value of public-domain software, and how it may evolve into ever more useful programs.

There is, however, a minor bug in the implementation of the "switch/case/default" triad. A construct such as

```
while ( *x )
        switch ( *x++ )
                {
        case A:     continue;
        case B: . . . . . ;
                }
```

will result in a jump to an undefined label, when the code for the "continue" statement is generated. What should happen is a branch to the loopback point of the enclosing "while" statement, (cf. Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, Appendix Z, section 9.9, Prentice Hall Software Series, ISBN 0-13-110163-3).

The fix involves minor changes to the calls to "addwhile" in "dowhile," and "docont," and a new function, "backq" (see Listing, page XX). This fix has the side-effect that a "continue" inside a "switch" which has no enclosing loop will generate the code for a "break," although the "no open loops" error will be generated.

I hope that this is of some interest.

Yours faithfully,
Andrew Macpherson
29 Tisbury Road, Apt. 7
HOVE
E. Sussex, BN3 3BJ.
England

EDITORIAL

As the 1983 issues of *Dr. Dobb's Journal* unfold, you may see a few slight cosmetic or content changes. One thing that you will notice as time goes on is that the listings we will be giving you will be shorter and meatier. The central ideas behind many listings may be condensed into exemplary pieces of code which may be more instructive. In addition, we hope to present more flowcharts and algorithms so that the ideas presented may be generalized more easily. When we do a condensation, we will also endeavor to make the complete listing available to those who wish a copy. We, of course, are still open to publishing longer listings if the listings are of sufficient import or if we deem it necessary in the interest of clarity. This is an attempt to get more editorial material to you, the readers, and we would love feedback on how well it serves your needs.

Another way in which we hope to serve you in the coming year is by continuing to publish top quality advertisements. We pride ourselves on the integrity of our advertisers, as well as our editorial-to-advertising ratio. In those very rare instances when an advertiser is providing poor services (or more importantly *no* service), we would like to hear from you as quickly as possible in order to investigate and take appropriate steps. Lately we have received some complaints (and registered some of our own) about a couple of our past advertisers. We regret this inconvenience to our readers, but it is impossible to screen out on the front end. While *caveat emptor* is still the rule of the marketplace, it helps to know where you can go for reputable purveyors of products and services. With your assistance, *Dr. Dobb's* will remain such a place.

Keep us posted!

Reynold Wiggins
Associate Editor

MODIFICATIONS TO SMALL C COMPILER FOR CORRECT 'CONTINUE'S

```
/*
** addwhile new nesting level
*/

addwhile ( ptr, switflag )
int ptr [], switflag;
{ int k;
          ptr [ WQSP ] = csp;
          ptr [ WQEXIT ] = getlabel ();
          if ( switflag == YES )
                    { int *p;
                    if ( p = backq ())
                              ptr [ WQLOOP ] = p [ WQLOOP ];
                    else      ptr [ WQLOOP ] = ptr [ WQEXIT ];
                    }
          else      ptr [ WQLOOP ] = getlabel ();
          if ( wqptr == WQMAX )
                    {
                    error ( "too many active whiles" );
                    exit ( ERRCODE );
                    }
          k = 0;
          while ( k < WQSIZ )
                    *wqptr++ =  ptr [ k++ ];
}
/*
** docont  jump to loopback point of innermost loop
*/

docont ()
{ int *ptr;
          if ( ! ( ptr = backq ()))
                    {
                    error ( "no open loops" );
                    if ( ! ( ptr = readwhile ())) return;
                    }
          modstk ( ptr [ WQSP ] );
          jump ( ptr [ WQLOOP ] );
}
/*
** backq new function : find the innermost loop on while queue
*/

backq ()
{ int *inner, *outer;
          if ( wqptr == wq ) return 0;
          inner = wqptr - WQSIZ;
          outer = inner - WQSIZ;
          while (( outer >= wq) &&
                   ( outer [ WQLOOP ] == inner [ WQLOOP ] ))
                    {
                    inner = outer;
                    outer -= WQSIZ;
                    }
          if ( inner [ WQLOOP ] == inner [ WQEXIT ]) inner = 0;
          return inner;
}
```

No BDOS Trace

Dear Doctor,

I have never been able to afford a subscription but do enjoy *DDJ* when I can borrow it from my friends. In the last issue so acquired (December, 1982) there was a letter from a person having trouble using a debugger to trace execution of a program through the CP/M BDOS.

This will never work no matter how clever one is with the break points because the CP/M BDOS is not re-entrant. On entry to the BDOS at 0005h, you jump to code which saves register arguments, the stack pointer, and the return address at fixed locations. The BDOS then switches to an internal stack for further operations.

Since DDT/SID/ZID use BDOS character I/O when displaying the trace information, attempts to trace through the BDOS will result in the saved values being overwritten and destroyed by the recursive BDOS call made by the debugger.

(Letters continued on page 74)

PISTOL

A Forth-like Portably Implemented STack Oriented Language

by Ernest E. Bergmann

Ernest E. Bergmann, Physics, Bldg. No. 16, Lehigh University, Bethlehem, PA 18015.

The development of PISTOL is based on STOIC, STack Oriented Incremental Compiler, a compiler developed by the Harvard and MIT Bio-Engineering center in 1977. STOIC is available from the CP/M Users Group as Volume 23a. Forth is available from the Forth Users Group (FIG), Box 1105, San Carlos, CA 94070. PISTOL is being provided in the public domain with this article. It may be reproduced and distributed for any and all uses so long as a proper copyright notice and acknowledgement of source is included.

PISTOL is available on Disk (see Figure 1, page 14) for $10.00 plus the price of the media, through the BDS-C User's Group, Box 287, Yates Center, KS 66783, and through the Amateur Computer Group of New Jersey (ACGNJ), Box 319, South Bound Brook, NJ 08880. – Ed.

PISTOL was developed with a different philosophy, perhaps, than its predecessors, Forth and STOIC. It is my belief that Forth-like languages should be available to users of large mainframe machines as well as mini and micro users. I have concentrated upon the following goals: user friendliness, portability between machines with different word-lengths and machine instruction sets, KISS ("Keep it simple stupid"), and freedom from many of the idiosyncrasies that have irritated me with Forth and, to a lesser degree, STOIC. Lastly, I wanted PISTOL to be as self-contained and complete as possible. Obviously, the result is not as chintzy about RAM and file space as its predecessors.

Before the Forth enthusiast is discouraged from examining this language further, I hasten to add that there are a number of simplifications that I have introduced which might be worth considering. I shall now detail some of the ways that PISTOL differs from classic Forth.

Following the lead of STOIC, PISTOL supports strings as a fundamental element of the language. They are manipulated with almost the same manner and ease as numbers. For example, to define a word, "DEMO," Forth starts off with:

```
: DEMO . . .
```

whereas STOIC and PISTOL start off with:

```
'DEMO : . . .
```

This could be very important if the choice of name is to be flexible. How would one write in Forth the definition for "3" depending upon whether the constant "FRENCH" were true or not?

```
FRENCH IF
    'TROIS
   ELSE
    'THREE
   THEN : 3 ;
```

Again, following the lead of STOIC, we compile everything into a compile buffer. Thus there is no interpret mode as in Forth. This change simplifies the language because we don't have to code two distinct modes and provide two separate sets of rules. To illustrate this point:

```
20 0 DO (whatever) LOOP
```

would work just fine typed in for immediate execution; it does *not* have to be embedded in a definition!

We differ from STOIC and early forms of Forth by storing the complete name of each definition. STOIC, for example, saves only the first five letters (and a letter count). This was done because it simplifies coding the language originally, the variable length of the name might actually take no more space, and, most importantly, when we use the disassembler and the trace features of PISTOL, we see the full, original names of the words used.

We have "packaged" PISTOL so that the disassembler, trace, and editor are always resident. (They probably could be removed, but why not enjoy the synergism of the complete environment at your fingertips!)

The prompt is much more informative than in other languages. STOIC started this trend by displaying "nesting depth." We have continued this trend by displaying the number of elements on the parameter stack when it is not empty. The prompt explicitly shows the current number base being used. If the nesting depth is not zero, one sees what one is "nested into." This information is used also for more vigilant syntax checking. I have worked with the earlier languages that fatally "bomb" with such lines as:

```
IF . . .;
```

or

```
DO . . THEN
```

We have been able to remove the "arbitrary" restriction that a ": . . . ;" definition can only be made at level 0. We can and do make extensive use of definitions that are nested inside of conditional expressions or, even, inside of other definitions!

We do not support "CODE" definitions. This is because we wish to guarantee portability of source code between widely differing machines. In addition, we wish to maintain our goal of user friendliness by being able to disassemble virtually everything. Obviously we depart from STOIC particularly on this point, but STOIC is not portable to non-8080-compatible machines.

Because the primitives of PISTOL are defined in some high-level language, such as Pascal or C, it is expected that additional primitives might be added to the "kernel" as the need arose. For example, in order to increase the speed of PISTOL for block move operations in editing, we might want to replace LDDR, a slow, PISTOL language definition. On a Z80 machine there is an obvious machine hardware instruction

Figure 1.

Contents of PISTOL Diskette

CABORT.ASM	code for MAC to make CABORT.CRL that contains the abort() function, similar to the setjmp() and longjmp() of BDS C v 1.45a
CABORT.CRL	
CORE1.	"core image" that results from compiling PBASE (see PISTOL.DOC for details)
ENVIRON.DOC	suggests at least 48K RAM total
PBASE.	PISTOL source code used to "educate" PISTOL.COM to be as smart as described in PISTOL.DOC
PISTB.C	module #2
PISTB.CRL	
PISTC.C	module #3
PISTC.CRL	
PISTD.C	module #4
PISTD.CRL	
PISTE.C	module #5
PISTE.CRL	
PISTF.C	module #6
PISTF.CRL	
PISTGEN.SUB	SUBMIT file to generate this disk
PISTOL.C	main module
PISTOL.COM	*** this and: 'CORE1 RESTORE will give you a new development language; try it, you'll like it (check the 40K documentation for best results!)
PISTOL.CRL	
PISTOL.DOC	40K documentation of PISTOL
PISTOL.H	constants and externals for PISTOL
PISTOL.PAS	42K Pascal version of PISTOL that runs on a DEC-20 mainframe computer; it helps define the way PISTOL should be created
PISTSUB.SUB	SUBMIT file to compile and link all C modules
READ.ME	History and editorial vis-a-vis with Forth and STOIC
SESSION0.DOC	example, chatty session showing some of the most basic operations available in PISTOL
SESSION1.DOC	more of the above, shows how CONSTANTs, VARIABLEs, new words and macros are created

that could be used instead. One could add this word to the kernel and "generalize" PBASE by replacing:

```
'LDDR : . . .
```

by

```
'LDDR DUP FIND  IF
                  DROP
                ELSE
                  : . . .
                THEN
```

To slightly ameliorate the lack of "CODE" definitions, we have provided an "inline macro" facility. If one creates a word definition using:

```
$ : . . . ;$
```

instead of:

```
: . . . ;
```

then when the word is encoded, the contents of the definition are compiled into the compile buffer instead of a "call" to the definition.

A number of conveniences are provided which are not common to the "competition." The crude line editor that is provided with PISTOL can be used in a BASIC-like fashion. In addition to being able to LOAD files, one can LOAD the edit buffer as well, starting at any particular line number. For example:

```
'MYWORK.PIS LOAD
```

and

```
3 LOAD
```

There are times when it is nice to record the terminal session. Instead of using a printer, one can create a disk file that will hold the record:

```
'XYZ LISTFILE
LIST ON
.
.     (this will be recorded)
.
LIST OFF
.
.     (this will not)
.
LIST ON
.
.     (this will)
.
etc.
.
.
BYE  (this way of exiting PISTOL will properly close
        all files that are opened for writing)
```

Undoubtedly, there are other important differences that I have not discussed; I hope that you will give PISTOL a try. I would be most pleased to receive your comments, constructive criticisms and questions. DDJ

DOBB'S EX MACHINA

by Hank Harrison

Anadex Has Competition!

At last Anadex, one of the most popular American-made dot matrix printer manufacturers, will probably have to bring their replacement ribbon prices down. Now Aspen Ribbons, Inc. of Boulder, Colorado, cuts that price by at least 50% at the retail level in quantities of one. For larger orders, the price goes down even further. The Aspen ribbons are manufactured in America, and no part of the ribbon is imported. Our testing of the two ribbons side by side found no significant difference, and the Aspen is slightly better in mechanical operations, bearing surface, etc. As with Anadex, one size fits the Plessy PM-LC11 and the entire Anadex range, 9000, 0991, 9500, 9501, 9620, 9625, and all DP series, the WP6000 and others. One Aspen-Anadex replacement, in quantities between 1 and 35 will cost $13.00 per unit, 144-287 units (wholesale) will cost $8.75 per unit, and any order above 864 units will be considered O.E.M. and will go out at $6.75 per unit. Color choices are available at slightly higher prices.

For further information, contact: Aspen Ribbons, Inc., 1700 N. 55th St., Boulder, Colorado 80301. Telephone (303) 444-4054, or toll free (800) 525-0646.

An Apple a Year . . .

All one has to do to cause Appleplexy at Apple these days is mention the word IIe. This is the code word for the latest Apple Surprise due any year now. By skillful use of VLSI technology, Apple has been able to design a machine that will please everyone interested in Apples as it looks externally like the old Apple.

Everyone knows that Apple is working feverishly on the LISA project, a 16- to 32-bit system that will display certain levels of artificial intelligence. The "Lisa" is the UNIX Mother Project that will hook up with the MACKINTOSH 68000 machine. Lisa and MackIntosh will not talk to Super IIe and III, but who cares? It is rumoured that MackIntosh will be portable, or at least portabalizable.

The Super IIe will be a boon and a Godsend and it will keep Apple fans in the fold. It will also contain a number of bells and whistles that perhaps should have been available in the past but which for various reasons were deemed unappleish. It now has upper and lower case, cursor controls in all directions, and 64K RAM standard with expandable memory architecture. The new system will have the 80 column capability, a paddle 0 and paddle 1 port enhancement, monitor support for HIRES graphics, auxilliary video with row addressing, a better and easier-to-use back panel with external connectors, a delete key, soft keys, relocatable French and German key caps and special International Key Board ROMs.

Full file compatability between Apple II and Apple III is assured and a true applications environment will be striven for through Profile, Duofile and Unifile, hard disk protocols. The floppies will have more capacity, possibly even a half-height profile, and the Z-80 card will continue as a co-processing option for CP/M devotees.

As to software, it appears that DOS 3.3 will be frozen but will continue up the scale with PRODOS which will be available in Spring, 1983.

Other features that are rumored to be included in the new Super Apple are Autostart ROM, Diagnostics, baud rate switching support and lots of software. We think the price should stay in the $2000 dollar range and we don't think Apple will get caught in a price war. Hats off to Apple on this one.

Tiny Scale Apple Polisher

Tom Crosley, the formidable multilingual consultant who developed the hugely successful PIE Writer way back when the 40-column board was the standard, is working on a new language so simple, it's elusive. It's one of those "wish I had thought of it" languages, the name of which I cannot divulge. It is, however, a small whopper – derived from PL/M, Pascal, and C, a portable language intended for both systems and applications programming on micros. The new language is tiny and is more effective as a whole than any of its component languages. We hope to feature it in *Dr. Dobb's* in its pre-release phase.

A Whale of an Assembler

David Eyes, head of technical product development at Hayden's new software think-tank in Lowell, Massachusetts, is rumored to be working on an amazing macro assembler package authored by a very talented new face from New Mexico. The new "low ticket" package will go on the block sometime in early 1983. It is dedicated to 6502 assembly language but the low ticket analogy ends there. The new package features forty monitor commands, more than fifty editor commands, a macro library, a subroutine library and a fully integrated architecture. The program utilities effectively utilize memory in a clever and never-before-used fashion so that the entire package can be thought of as a workbench system with tools. This macro assembler can, in edit mode, access a string buffer, a character buffer, and an on-screen text buffer. It comes with an extensive documentation package. The product should surface for air soon and *Dr. Dobb's* will be reporting on it.

Semi-Disk Systems

I hate semi-disk systems! I hate anybody whose hardware works the first time. I hate them too, because they beat me to a logo device – the universal sign of negation. I wanted it for use with a book cover, the title of which was *Not Dead*, but they can have it because their board works better than my book.

Anyway, the 500K semi-disk, in the S-100 form, is the neatest board job I've ever seen. It ran a 115K Wordstar text file using control QR from head to foot in 12 seconds on an IBM PC at Comdex. Very impressive. My Victor 9000 does the same thing in a minute-and-a-half, given the disk access using the 8088 and 126K. The semi-disk makes an 8-bit act like a 16-bit machine.

As currently supplied, the semi-disk provides an optional battery back-up which can protect data in RAM for ten hours. Semi-disk is available for TRS-80, S-100, and the IBM PC. A full and exhaustive hardware review and shakedown will be presented in *DDJ* soon.

MCDISPLAY

MCDISPLAY is an interface module for high-level languages running on any Z-80, 8080 or 8085 system. It requires 54K of memory and provides a very simple, compact and elegant way of interactive communication. Version I supports Microsoft BASIC (both the version 5.2 interpreter and the version 5.3 compiler). Modules for other languages will follow.

Programs designed with the MCDISPLAY runtime module are organized around the display. This method provides the programmer with the capability of complete and controllable access to the user without having to write large a-

(Continued on page 73)

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

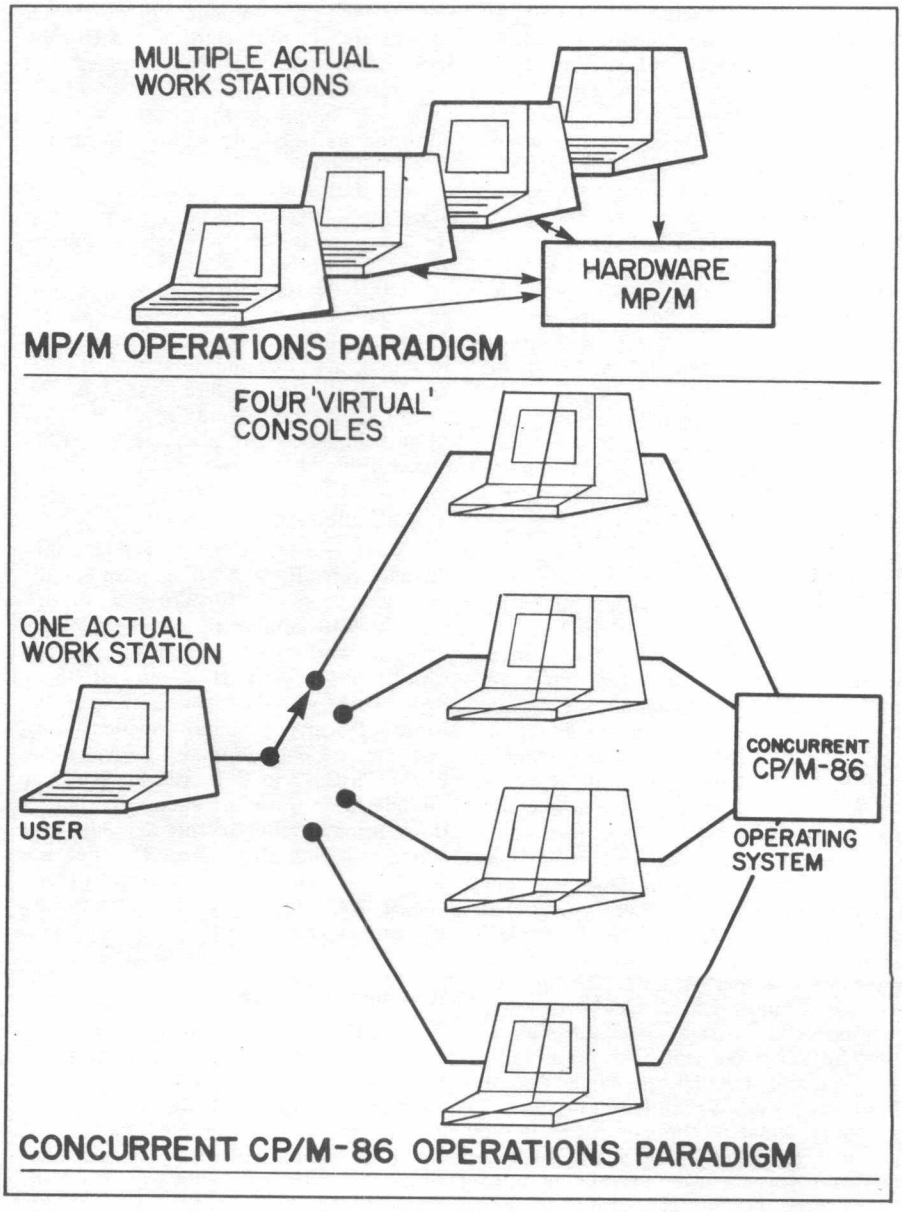
This month I will devote the column to a review of Concurrent CP/M-86 (CCPM), which has recently been released by Digital Research for the IBM Personal Computer. The CP/M-86 operating system was customized and marketed for the PC as an IBM software product, but apparently Digital Research was displeased with the result and undertook to implement and sell the Concurrent version directly. This is fortunate from the standpoint of evaluating the operating system's performance, since it gives us an "official" version which has been presumably optimized and polished to the complete satisfaction of its inventors – we don't need to concern ourselves with the hardware-dependent foibles that sometimes creep into OEM implementations. Versions of Concurrent CP/M-86 for other 8086/88 microcomputers will be available eventually, but will probably be many months in arriving.

A Little Background Information

For those of you who are not familiar with the Digital Research family of operating systems, a brief overview may be in order. The original CP/M, now known as CP/M-80, is a single user operating system for 8080/Z-80 microcomputers that provides hardware-independent console, printer, and file handling services for application software and program development tools. CP/M-80 has become the industry standard for 8-bit microcomputers. There are several hundreds of thousands of licensed installations, and an unknown but probably also enormous number of unlicensed copies in circulation (as well as a large number of licensed users of "CP/M compatible" operating systems such as CDOS, SDOS, and TurboDOS). The third major revision of CP/M-80 has just been released, with an improved file manager, enhanced utilities, and provision for powerful graphics extensions.

The multi-user version of CP/M-80 is known as MP/M-80, and is now in its second major release. The original version of MP/M had irksome deficiencies in performance, was difficult to configure, and included no provision for file or record locking so that the integrity of data could be protected when several programs were executing simultaneously. These problems were virtually eliminated in MP/M-80 version II, which is a very acceptable real-time, multi-tasking operating system; however, it is inherently limited by the speed and power of the host 8-bit microprocessor and cannot practically support more than about four consoles in standard systems.

CP/M-86 is a translated version of CP/M-80 for the Intel 8086/88 family of microcomputers. From the user's point of view, operation of CP/M-86 is extremely similar to that of CP/M-80, with most of the same commands. Disk allocation and file management are identical to that of CP/M-80, so that data may be easily transported between programs running on either operating system. CP/M-86 has been a fairly stable operating system since its release, and has not required any major revisions. The performance of CP/M-86 has been limited somewhat by the constraint of compatibility with the 8-bit systems, and it has run into stiff competition for the hearts of microcomputer manufacturers and users from the Micro-



soft counterpart "MS-DOS."

Finally, MP/M-86 is available as the upward-compatible, multi-user version of CP/M-86. This is an exceedingly powerful microcomputer operating system readily capable of supporting up to sixteen users. It offers many additional sophisticated services to the programmer such as queue management, dynamic memory allocation, easy installation of resident system processes, multiple printer support, and task spawning. It requires a comparatively large amount of memory (256 Kbyte RAM systems are common) and is usually hard-disk based. Efficient implementations of MP/M-86 with proper interrupt handling and buffered I/O are quite complex and require several man-months of work. For this reason, it has been slow to appear on the market for the various 8086/88 microcomputers. An unusual variant called MP/M 8-16 is sold by G&G Engineering for the CompuPro 8085/8088 dual processor board. It allows the interleaved execution of programs coded for the 8080 or 8086 on the same microcomputer without any special intervention by the user.

What is Concurrent CP/M-86?

To quote Digital Research, "Concurrent CP/M-86 is a single-user, multitasking operating system that lets you run multiple programs simultaneously by dividing tasks between virtual consoles." Inspection of the documentation, services, and utilities of CCPM reveals it to be a slightly stripped-down version of MP/M-86, with some rather novel modifications. From the operating system's perspective, there are four user consoles which can originate and interact with executing processes. From the user's point of view, there is one physical console which serves as a "window" and can be switched at will between any of four virtual consoles (see Figure 1).

The terminal handlers for the four virtual consoles buffer output independently; when a program writes to a virtual console that is not currently being viewed by the user, the characters are stored into a RAM swapping buffer and optionally spooled into a disk file so that no output is lost. When that virtual console is finally selected by the user, the saved output is recalled from the swapping buffer and displayed on the screen. Toggling the physical console between the four virtual consoles is accomplished merely by holding down "Control" and keying a number between 0 and 3; the response by the operating system is instantaneous. While the system is running, the status line at the bottom of the screen displays the following information for the currently selected virtual console: the name of the active task, the unit number of the attached printer, names of disk drives with any open files, the time, and the status of certain keys such as CAPS LOCK and NUM LOCK.

Table 1.

Concurrent CP/M-86 commands and utilities.

Command name	Action
ABORT	Stops execution of a task.
ASM86	8086/88 assembler.
CONFIG	Sets operating parameters for serial ports.
DDT86	8086/88 interactive debugger.
DIR	Display disk directory.
DSKMAINT	Initialize or copy disks.
DSKRESET	Log in diskette.
ED	Line editor.
ERA	Erase file(s).
ERAQ	Erase file(s) with query.
FUNCTION	Program special function keys.
GENCMD	Create executable file from assembler output.
HELP	On-line assistance for system commands.
PIP	Transfer and manipulate files among other things.
PRINTER	Select or display attached printer number.
REN	Rename file(s).
SDIR	Sorted directory listing with many options.
SET	Controls password protection and file attributes.
SHOW	Displays disk and system parameters.
STAT	Display file information, assign attributes.
SUBMIT	Batch processing utility.
SYSDISK	Designate the "system" disk drive.
TOD	Set or display time and date.
TYPE	Display contents of a text file.
USER	Select or display the current user number.
VCMODE	Set mode and buffer size for virtual consoles.

System Requirements

The power of Concurrent CP/M-86 carries a hefty price tag for both hardware and software. The first requirement, of course, is that you fork over $350.00 to Digital Research for a single user license.

Next comes the hardware. A minimum of 256 Kbytes of RAM is recommended by Digital Research; at least 192 Kbytes must be present in order to boot up the system. If more memory is available, part of it can be allocated as a "virtual disk."

Two disk drives are mandatory; double-sided drives make the system much more pleasant to use due to the large amount of disk storage occupied by the various system utilities and the HELP text file.

Commands and Utilities

Users of CP/M-80 and CP/M-86 will find much that is familiar. The well-known commands DIR, ED, ERA, PIP, REN, STAT, SUBMIT, TYPE, and USER that hearken back to the early days on the 8080 operate in the same old tried-and-true manner.

The utilities ABORT (terminate executing task), DSKRESET (log in new diskette), ERAQ (selective file erase with query), PRINTER (select list device), SDIR (formatted directory with file sizes and attributes similar to STAT *.* or LST), SHOW (display system and disk parameters), SET (control disk and file attributes, passwords, and time stamping), and TOD (set or display system time and date) work in the same manner as their counterparts on MP/M systems.

The program development tools ASM86 (8086/88 assembler), DDT86 (interactive debugger), and GENCMD (create

executable command file) are identical to their CP/M-86 counterparts.

Finally, there are a number of utilities which are unique to the IBM Personal Computer implementation of Concurrent CP/M-86. CONFIG allows the user to control the baud rate, word length, parity, and stop bits of the serial ports. DSKMAINT provides formatting and copying of single- or double-sided diskettes. FUNCTION allows the user to specify character strings for each of the programmable function keys. HELP provides interactive, on-line explanations and examples for any of the system commands. SYSDISK allows the user to specify which disk drive will be searched for a program if it is not found on the "current" or default disk. Finally, VCMODE is used to control the characteristics of each of the virtual consoles. See Table I on page 21 for a more condensed list of the system commands.

File Handling

Concurrent CP/M-86 incorporates the Digital Research BDOS (Basic Disk Operating System) version 3, which is also embedded in MP/M version II and supports many advanced file management options. This BDOS apparently will become the standard for all of the DRI operating systems including the new CP/M-80 version 3.0.

On the command level, diskettes and individual files can be protected by passwords. Time stamping for the creation, update, or last access of a file is supported. In addition, files may be marked with the attributes SYSTEM, READ-ONLY, and ARCHIVED.

On the programming level, executive service calls available to application programs provide full support for file and record locking. If a file is opened in "unlocked" mode by a program, it remains accessible to any other task which is also willing to open it in "unlocked" mode. A file can be opened and shared in "read only" mode by any number of concurrent processes. If a program wishes exclusive access to a file, it can open it in "locked" mode; the operating system will deny any requests by other tasks to open the same file.

When two or more processes share a file in "unlocked" mode, both having the privilege of write access, there is a potential danger of loss of information. Consider the following situation: task #1 reads a record from the disk into memory. By coincidence, task #2 reads the same record immediately thereafter. Task #1 makes some changes to the record and writes it back to disk. Task #2 makes some different modifications to the data and writes it to the disk. Depending on which task happens to get serviced for its disk write request first, one or the other set of changes will be permanently lost. In addition, one of the tasks may take some erroneous action based on the contents of the disk record which was not yet updated by the other task.

The capability called "record-locking" is designed to forestall such disasters in a multi-tasking environment. When a program issues a read request for a record, it can optionally specify that the record should be "locked," that is, the record is made unavailable for updating by any other process until it is released by the program that issued the "lock" request.

Table II

Concurrent CP/M-86 function calls. Where I have designated console functions as "compatible" with CP/M-86 or MP/M, I am ignoring issues of "virtual" versus "physical" consoles. Where I have called file and record functions "compatible" with CP/M, I am assuming an application running as the sole task and disregarding problems of file or record sharing/locking.

Function		Action	Function		Action
0	%	System reset—terminate calling process	26	+	Set memory address for disk transfer
1	+	Console input	27	+	Get address of disk allocation table
2	+	Console output	28	+	Write protect disk
3	#	Raw console input	29	+	Get disk read-only vector
4	#	Raw console output	30	*	Set file attributes
5	+	Output to default list device	31	+	Get address of disk parameter block
6	*	Direct console I/O	32	+	Set/get user code
7	#	Get I/O byte (not implemented)	33	#	Read random record
8	#	Set I/O byte (not implemented)	34	#	Write random record
9	+	Print string	35	+	Compute file size
10	+	Buffered console input	36	+	Set random record
11	+	Read console status	37	+	Reset disk drive (to not logged-in state)
12	%	Return BDOS version number	38	#	Access drive (place on lock list)
13	%	Reset disk system	39	#	Free drive (remove from lock list)
14	+	Select disk	40	*	Write random with zero fill
15	+	Open file	41	#	Test and write record
16	+	Close file	42	#	Lock record
17	+	Search for first directory match	43	#	Unlock record
18	+	Search for next directory match	44	#	Set multi-sector transfer count
19	+	Delete file	45	#	Set BDOS error mode
20	+	Read sequential	46	#	Get amount of free disk space
21	+	Write sequential	47	#	Chain to program
22	+	Make file	48	#	Flush buffers
23	+	Rename file	49		(not implemented)
24	+	Return log-in vector	50	*	Direct BIOS call
25	+	Return current disk	51	*	Set segment address for disk transfers

This allows a record to be safely modified by one program without fear of interference by other tasks that may be executing simultaneously. As an alternative to file and record locking (which consumes system resources and can slow down the system), a "Test and Write Record" function is provided which verifies that the disk copy of the record is unchanged before honoring a request to update it.

Additional file security is provided by extensive checksum verification of disk directory entries and all active file control blocks. This is designed to prevent any compromise of data integrity by a task which has crashed, run out of control, or is performing disk access in a non-standard manner. It should be noted that some techniques of file access which were legal under earlier versions of CP/M, especially those involving direct manipulation of the file control block or simultaneous access to several different file "extents," are intercepted and aborted by Concurrent CP/M-86 and the other operating systems which use the third generation BDOS.

Even with the safeguards described above, concurrent access to multiple files by more than one program is tricky business. It requires careful analysis and a programmer experienced with multitasking operating systems in order to be done safely. For example, it is quite easy to have two programs get into a "race condition" where each has locked a record that the other needs, and thus neither one can continue to execute—system resources are tied up irrevocably and usually the operating system will slow down and "die."

Executive Services

Concurrent CP/M-86 offers a multitude of function calls for use by the application programmer; a complete list of the Concurrent CP/M-86 operating system services is given in Table II.

In addition to the usual functions 1-40 which are present in ordinary CP/M systems, there are entire new repertoires of capabilities which are highly reminiscent of a powerful minicomputer real-time operating system such as DEC's RSX-11M. For example, functions 53 through 58 and 128 through 130 allow a program to dynamically request, use, and release memory from the system pool based on requirements determined at run-time. Functions 131 through 133 allow peripheral device drives to communicate with other tasks through use of "semaphores" or software-controlled flags.

Functions 134 through 140 allow tasks to create and maintain "queues," which can be used to pass information asynchronously between processes. Functions 146 through 149 and 158 through 162, among others, allow a given program to attach or detach itself for input and output from any of the virtual consoles or printer devices. Functions 150 and 152 assist an application program in parsing filenames and interpreting command lines. Lastly, function 144 allows a task to initiate or "spawn" other processes which can run concurrently.

An interesting observation in passing is that BDOS function 12, which fetched a simple one-byte operating system version number in CP/M-80, has been drastically extended. It now returns a sixteen-bit parameter that is broken up into a number of bit fields to specify the BDOS

(Table II continued)

Function		Action
52	*	Return segment address for disk transfers
53	*	Get largest available memory region
54	*	Allocate maximum memory at absolute address
55	*	Allocate memory segment
56	*	Allocate memory segment at absolute address
57	*	Free memory segment
58	*	Free all memory segments
59	*	Program load
100	#	Set directory label
101	#	Return directory label
102	#	Read extended file control block
103	#	Write extended file control block
104	#	Set date and time
105	#	Get date and time
106	#	Set default password for file access
107	#	Return serial number
128	#	Memory segment allocation
129	#	Memory segment allocation
130	#	Free memory segment
131	#	Poll device (test logical interrupt flag)
132	#	Wait for a system flag
133	#	Set a system flag
134	#	Create system queue
135	#	Open queue
136	#	Delete queue
137	#	Read message from system queue
138	#	Conditionally read message from queue
139	#	Write message into system queue
140	#	Conditionally write message into queue
141	#	Delay for a specified number of clock ticks
142	#	Call system dispatcher
143	#	Terminate calling process
144	#	Create (spawn) a process
145	#	Set priority of calling process
146	#	Attach console
147	#	Detach console
148	#	Set default console number
149	#	Assign console to another process
150	#	Interpret and execute a command line
151	#	Call function in resident procedure library
152	#	Parse filename
153	#	Return number of default console
154	#	Get address of System Data Area
155	#	Get date and time
156	#	Return address of process descriptor
157	#	Abort specified process
158	#	Attach list device
159	#	Detach list device
160	#	Select default list device
161	#	Conditionally attach list device
162	#	Conditionally attach console device
163	%	Return Concurrent CP/M-86 version number
164	#	Return number of default list devices

\+ = compatible with CP/M-80, CP/M-86, and MP/M
* = compatible with CP/M-86 and MP/M
\# = compatible with MP/M-86
% = similar to CP/M-86 or MP/M, but with subtle differences

version, the environment (whether CP/M, Concurrent CP/M, MP/M, or any of the preceding with networking), and the CPU type. Digital Research is evidently planning well ahead here. The day is not far off when the entire DRI family of compilers and operating systems will be running on the 8080, Z-80, 8086/88, 68000, Z8000, and 16000 microprocessors.

Noteworthy Features

As we have already seen, Concurrent CP/M-86 contains much that is new and different from the CP/M systems we are all accustomed to, but there are a few features which are especially welcome and should be mentioned here.

- File access has been enhanced with a special "burst mode," which allows the transfer of one to sixteen records with a single function call.
- A complete set of BDOS disk error codes are defined, and an application program may choose to handle all physical and logical disk errors. This makes it possible for editors, disk utilities, and other critical programs to handle formerly fatal events (such as the disk drive door not being closed) in a graceful way. The dreaded message, "BDOS ERROR ON XX," which has been the cause of so much cursing by users and programmers alike, can now be eliminated forever.
- The new system utilities, DSKMAINT, CONFIG, and FUNCTION, are screen-oriented, menu-driven, make extensive use of the programmable function keys and video attributes, and are practically idiot-proof. I hope this is a sign of the trend to be taken by future Digital Research products.
- A "virtual disk" capability, sometimes known as "RAMDISK" or "M-DRIVE," is built into Concurrent CP/M-86. If the operating system finds that the system has memory installed at the proper address, the virtual disk is automatically made accessible as drive "M:". It can be used to store the commonly needed system utilities such as PIP and STAT and any temporary files created by SUBMIT, improving the perceived responsiveness of the system dramatically.

Documentation

The manuals for Concurrent CP/M-86 demonstrate a quantum jump in usability as well as flashiness. The documentation is divided into the 204-page *User's Manual* which describes operation of the various common system commands and utilities, and a 338-page *Programmer's Guide* which covers the file system BDOS function calls, assembler, and debugger in detail. Colored text is employed to emphasize example of actual system interactions. Liberal use of tables, diagrams, and appendices make information easy to locate. The manuals are packaged together in a hardboard, three-ring binder with accompanying box cover that is slightly larger than the IBM format.

As is usual for Digital Research documentation, the manuals are aimed at a relatively sophisticated reader, but they are well organized and lucid. There are a vanishingly small number of typographical and grammatical errors. In fact, the only technical error I have found so far that might affect a programmer is in Appendix H of the *User's Guide*, and involves an overlap between the video escape sequence for "Erase Entire Line" and "Insert Blank Line." Clearly, the new department Digital Research set up to enforce standards of documentation quality has made its presence felt.

What Price Concurrency?

With 320 Kbytes of main memory, I have found it possible to run three 64-Kbyte tasks simultaneously. This implies that the operating system with its various buffers requires about 128 Kbytes of RAM. The load image "CPM.SYS" on the disk drive occupies 92 Kbytes.

Naturally, operating system support for concurrent task execution entails a certain amount of overhead for servicing the system clock, maintaining various queues and lock lists, saving task contexts, and dispatching tasks based on their priority and demand for system resources. To try to assess the actual effect of this overhead on task execution time, I ran a well-known Forth prime number sieve benchmark program on the conventional IBM implementation of CP/M-86 and on Digital Research Concurrent CP/M-86. This is a classic "CPU-bound" program, with no disk I/O and only a small amount of terminal I/O. The program completed in 27 seconds on ordinary CP/M-86, and in 28 seconds on the Concurrent version (with no other active tasks)—approximately a 4% decrease in throughput. It is interesting to note that this benchmark under Microsoft PC-DOS requires only 25.7 seconds; the difference can probably be attributed to the time which is wasted updating the status line display on IBM's CP/M-86.

Then I ran the same CPU-bound benchmark on Concurrent CP/M-86 as two simultaneous tasks. They each completed in 56 seconds, exactly twice the amount of time required when the benchmark executed as a single task. This implies that the operating system overhead to support concurrency is relatively independent of the number of processes that are active.

To assess the effect of concurrent tasks on disk performance, I coded a small routine which performed 100 random reads of 1-Kbyte records in a 50-Kbyte contiguous data file. This is a truly disk-I/O-bound benchmark, since it performs almost no computation at all (except for the random number generator) and simply requests one disk access after another. On IBM's version of CP/M-86, the program completed in 75 seconds —a surprisingly poor performance. On Concurrent CP/M-86, with one task running, the benchmark executed in 52 seconds, approximately a 30% improvement! However, when the system was really put to the acid test with two copies of the disk-bound benchmark running simultaneously (accessing the same file on the same drive), its performance really fell apart and each task required about 430 seconds to complete. This seemed to be largely due to a large amount of disk "thrashing" that took place as the two tasks requested records in different extents, forcing many inspections of the disk directory.

I expect that the most common use of multi-tasking by the average user will be with one CPU-intensive program such as a spreadsheet or word-processor executing concurrently with spooling of a file or some other peripheral-bound process. Under such circumstances, the decrease in responsiveness that can be attributed to multi-tasking is practically unnoticeable. To verify this impression, I ran the two previously described types of benchmarks simultaneously. The CPU-bound program completed in 32 seconds and the disk-bound routine finished in 59 seconds, each showing only 14% degradation over the tasks executing alone.

A Few Small Gripes

In using Concurrent CP/M-86 for one month, I have not found any earth-shaking bugs or omissions. There are a few minor annoyances which I have taken the liberty of itemizing below.

There is no check for the disk drive door being closed; if a diskette is not loaded into a drive when it is accessed, the system will hang indefinitely without an error message.

There is no provision for accessing the bit-mapped graphics capabilities of the PC, even at the minimal level of simply plotting points. Control of cursor position, character attributes, selective erasing, and foreground and background colors in text mode *is* provided by means of escape sequences—very nice, but for unclear reasons the character underlining capability fo the video interface was omitted completely. Furthermore, Digital Research did not choose the same escape sequences for screen control as were used in the IBM version of CP/M-86, so that vendors of program packages will have to maintain and sell two separate versions.

Output to the video screen is inexplicably slow. Text appears to write on the screen at the equivalent of somewhere

around 4800 baud. I realize that there is unavoidable overhead involved in buffering output for four virtual consoles. Still, on such a powerful machine with memory-mapped video to boot, one might hope for a somewhat better effective speed of display.

The state of the CAPS LOCK, NUM LOCK, and WRAP parameters is not maintained individually for each virtual console. This is a minor point, but when you are flipping back and forth between an editor and some other utility, it is a nuisance to have to toggle these keys when the operating system could just as easily keep track of them for you.

The 8086/88 assembler does not include the control mnemonics for the Intel 8087 numeric coprocessor. Although they can be implemented by the user with the CODE-MACRO facility, this is a very laborious process.

The system boot file, startup batch files, and all of the commonly used system utilities will fit nicely on a double-sided disk in drive A, making the two-diskette system distributed by Digital Research unnecessary. But even if everything the system needs is on drive A and no auto-start batch files are present, it will still try to access drive B during the cold boot sequence. I have found no way to configure the system so that it will leave drive B alone.

Summary

Concurrent CP/M-86 provides the power to implement applications on a desktop machine that would have been unthinkable even two or three years ago. Even for the casual user, the flexibility and conviences that will result from the ability to execute several programs concurently is going to cause a rapid "revolution of rising expectations" that will have to be answered by the other manufacturers of microcomputer operating systems. I rate this software "Excellent."

DDJ

Program Linkage by Coroutines—Forth to BASIC

by L. Morgenstern, M.D.

Leonard Morgenstern, M.D., 304 Rheem Blvd., Moraga, CA 94556.

Interactive languages, such as BASIC, normally respond to commands typed by the operator. However, there are situations in which it might be desirable for the commands to be issued by a program, in effect making BASIC into a slave of that program. This can be accomplished by incorporating coroutines in the I/O. The method is a general one, and will be illustrated by the linkage of Forth and BASIC, in which Forth is the driver and BASIC a file handler and floating-point calculator.

The particular application that I had in mind was graphics display for a data base in microsoft BASIC. Forth seemed ideal for the purpose, but I did not want to lose BASIC's file-handling capabilities and high-precision floating-point package. There was also the important consideration that the data base program was already written.

The problem would be solved if I could find a way to get BASIC to accept commands issued by Forth. Then Forth could load and run BASIC programs. These would POKE results into memory where they would be available to Forth.

I was unable to work out a method, until I saw David Shaw's note on coroutines (*DDJ*, 63:4, January 1982). Using that principle, I solved my problem in a few hours. The actual implementation required only a few lines of code incorporated in the I/O segment of each program.

In this paper, I will describe the general principle, followed by an outline of the specific application.

Principle of Operation

A coroutine can be thought of as a separate program module that is callable somewhat like a subroutine. Each coroutine has its own stack. When a coroutine is called, the coroutine handler pushes the return address onto the current stack, stores the stack pointer in an array, and replaces it with the new one. Like a subroutine, a coroutine may be called from anywhere in memory, and the return will be to the memory address immediately following the call. However, a return is not accomplished by a RET command, but rather by calling the original program as if it were another coroutine. It is not necessary to return directly (or for that matter, ever) to the most recent calling program. Therefore, coroutines do not need to be nested, as do subroutines. Coroutines are useful in multi-programming, interrupt handling, and complex I/O manipulation.

In the method used here, up to 128 coroutines may be defined, each assigned a number divisible by 2. Coroutine n is called by the sequence,

```
MVI A,n
CALL CSWAP (see below for
    definition of CSWAP)
```

The coroutine handler has two entries, CBEGIN and CSWAP. (I have used these names to avoid confusion with the Forth words, BEGIN and SWAP.) CBEGIN is the startup entry. CSWAP is called during program execution. It stores the stack pointer of the current program, and substitutes that of the one being called. Any registers that need to be stored must be pushed prior to calling CSWAP, and popped on return.

Only two coroutines are required for the application described in this article. Coroutine 0 will be a call from Forth to BASIC, and coroutine 2 will be a call from BASIC to Forth.

Linkage is accomplished by incorporating a coroutine call in the I/O segment of each program. Emission of data by Forth is handled by means of the code shown in Listing 1. The byte on the top of the stack is popped into register E, and then register E is moved to a memory location named CHAR. Control is then transferred to BASIC (actually to the North Star DOS) via coroutine 0.

BASIC, looking for input, moves CHAR to the A register, and tests it (Listing 2). It is non-zero. CHAR is now set to zero, and control returns to BASIC with the most recently transmitted character in the A register.

When BASIC needs input again, it finds that CHAR is zero. Control is switched to Forth via coroutine 2, where it stays until Forth is ready to output another character.

Implementation

My system is a SOL with North Star disk drives, using release 5.2 DOS, in which DOS and BASIC can be relocated to any address. By careful assignment, it is possible to get all of the necessary programs into memory at the same time.

0 -7FFF	FORTH
8000 -BFFF	BASIC
C000 -CFFF	COMPUTER OPERATING SYSTEM
D000 -DFFF	UTILITY AND SCRATCH AREA
E000 -E7FF	NOT USED
E800 -EFFF	DISK CONTROLLER
F000 -FCFF	DOS
FD00-FFFF	COROUTINES

The actual implementation required the following steps:

1. Assembling the coroutine handler

No essential changes were necessary to Shaw's program, but I have simplified his labeling scheme (Listing 5). The handler is prefaced by a jump table so that the entry CBEGIN is at ORG, and the entry CSWAP is at ORG+3. The parameter STAKSIZ sets the amount of memory to be allocated for each stack. Since both BASIC and Forth set their own stacks, it can be set to a low value. The parameter GO indicates in which coroutine the program will start, in this case 0.

Stacks are defined by two lines of code for each coroutine. The first is a DS statement that allocates room for the stack, followed by a DW statement labeled TnSTRT which contains the startup address for the coroutine. In the application illustrated in this paper, two coroutines are defined, labeled T0STRT and T2STRT. Next, a stack table is defined at STAKTAB. Its first entry is T0STRT, the second T2STRT, etc.

2. Modification of North Star DOS

The I/O segment of the North Star DOS needs to be rewritten to incorporate the code in Listing 2.

3. Loading Forth definitions

The Forth screens labeled 100 and 101 (Listing 3) define Forth words that incorporate the code in Listing 1. Screen 100 defines various addresses as constants, and also defines CO-OUT, a Forth word that incorporates the output code in Listing 1. The CFA (Code Field Address) of EMIT is also defined as a constant named EADDR. EMIT is the Forth word that sends characters to the console.

In screen 101, the words [CO and CO] are defined. [CO inserts the CFA of CO-OUT into the CFA of EMIT, thus diverting subsequent output to the co-

routine. CO] replaces the original value of the CFA. These words also set and reset, respectively, the two flags ISIG and OSIG. ISIG, when set, forces the North Star DOS to accept input from the coroutine (see Listing 2). OSIG suppresses all DOS output. This is desirable as DOS and BASIC would otherwise echo all commands. These flags are under program control, and can be overridden when necessary.

The word [CO causes most Forth output words, such as EMIT . .", etc. to send output to the North Star DOS via CO-OUT. An exception is CR which, in my system, causes a hang-up, the reasons for which are unclear to me. Instead, I use COCR, defined in screen 101.

4. Starting the program

To start, simply jump to CBEGIN. Program control will be assumed by Forth.

5. Sample program

In screen 102 (Listing 3), an example is given. It defines a Forth word LOADXYZ which will load a BASIC program named XYZ and then transmit the number 5 to it. The program XYZ (Listing 4) simply asks for the input of a number, and inserts it at RAM address 52224 (D000 hex). Control is transferred back to Forth, which sets the number from that memory address and prints it. ▶▶J

(Listings begin on page 28)

Listing One (Text begins on page 26)

Skeletal Output Module for Coroutine

```
BYTE TO BE TRANSMITTED HAS BEEN POPPED INTO
THE E REGISTER


LXI H,CHAR
MOV M,E          ; Move byte in E to CHAR
PUSH B           ; B reg must be saved in FORTH
MVI A,0          ; Coroutine 0 calls BASIC
CALL CSWAP
POP B
JMP NEXT         ; Back to Forth
```

Listing Two

Input Module for Coroutine

```
;SKELETAL LISTING FOR N* DOS INPUT ROUTINE
;
ISIG    EQU 0F9FDH          ; 0=normal, 1=coroutine
OSIG    EQU 0F9FEH          ; 0=normal, 1=suppress
CHAR    EQU 0F9FFH
CSWAP   EQU 0FD03H
;
;CIN1 IS NORTH STAR DOS INPUT ROUTINE
;OBTAINS CHARACTER VIA COROUTINE IF ISIG NOT 0
;OTHERWISE FROM KEYBOARD
;CHARACTER IS RETURNED IN A REGISTER

CIN1    LDA ISIG
        ORA A
        JZ KEYIN         ; If ISIG=0 then keyboard
GETC1   LXI H,CHAR
        MOV A,M          ; Move CHAR to A-register
        MVI M,0          ; Reset CHAR to 0
        ORA A
        RNZ              ; Exit if there was a byte
        MVI A,2          ; Otherwise, to FORTH via
        CALL CSWAP       ;    coroutine 2
        JMP GETC1        ; And try again, it's there now


KEYIN   ....             ; Keyboard input routine here
        ....
        RET
```

Listing Two (Continued)

```
;COUT IS NORTH STAR DOS OUTPUT ROUTINE.
;NORTH STAR ENTERS WITH A 'DEVICE' NUMBER IN REGISTER A,
;   BYTE TO BE EMITTED IN REGISTER B. ON EXIT, A MUST CONTAIN
;   A COPY OF B.
;OSIG CONTROLS ECHOING OF COMMANDS BY BASIC
;IF OSIG=1 THEN OUTPUT IS SUPPRESSED


COUT    PUSH PSW          ; Save device number on stack
        LDA OSIG
        ORA A
        JZ COUT1          ; If OSIG is 0 then normal output
        POP PSW
        MOV A,B           ; N* DOS requires this
        RET               ; If OSIG is non-0 then no output
;
COUT1   POP PSW
        ...               ; Regular output routine here
        ...
        MOV A,B           ; N* DOS requires this
        RET
;
;INITIALIZATION
TINIT   XRA A
        STA CHAR           ; Set CHAR and flags=0
        STA OSIG
        STA ISIG
        RET
```

Listing Three
Forth Definitions

```
( SCREEN 100)

HEX
FDOO CONSTANT CBEGIN        ( STARTUP ADDR FOR COROUTINE)
CBEGIN 3 + CONSTANT CSWAP   ( CSWAP ADDR FOR COROUTINE)
F9FF CONSTANT CHAR          ( ADDR OF CHARACTER TRANSMITTED)
F9FE CONSTANT OSIG          ( ADDR OF OUTPUT SIGNAL)
F9FD CONSTANT ISIG          ( ADDR OF INPUT SIGNAL)

CODE CO-OUT
     D POP CHAR H LXI E M MOV B PUSH
     0 A MVI CSWAP CALL B POP NEXT JMP C;

  EMIT CFA CONSTANT EADDR       ( CFA of EMIT )
```

(Continued on next page)

Listing Three (Listing continued, text begins on page 26)

Forth Definitions

```
( SCREEN 101)

: [CO    ( Insert CFA of CO-OUT into CFA of EMIT; set flags)
     [ ' CO-OUT CFA ] LITERAL EADDR ! 1 ISIG C! 1 OSIG C! ;

: CO]    ( Restore CFA of EMIT to orig value; reset flags )
     [ EADDR @ ] LITERAL EADDR ! 0 ISIG C! 0 OSIG C! ;

: COCR   OD EMIT ;          ( Use instead of CR )

: ^C     3 EMIT ;           ( Sends ctl-c to North Star DOS )



( SCREEN 102  EXAMPLE )

( ASK NORTHSTAR DOS TO LOAD BASIC PROGRAM XYZ AND RUN IT)

HEX
: LOADXYZ   [CO   ." GO BASIC " COCR
                  ." LOAD XYZ " COCR
                  ." RUN " 5 , COCR CO] ;
LOADXYZ D000 C@ .    ( Should print 5 )
```

Listing Four

Sample BASIC Program

```
10 REM SAMPLE BASIC PROGRAM XYZ
20 REM INPUT A DIGIT AND POKE IT INTO D000 HEX
30 INPUT I
40 FILL 52224,I
50 END
```

Listing Five

Coroutine Handler

```
                ;
                ;COROUTINE HANDLER. FOR FULL DOCUMENTATION,
                ;SEE TEXT AND DDJ 63:8, JANUARY 1982
                ;
FD00                    ORG     OFD00H
                ;
                ;JUMP TABLE
FD00 C32CFD             JMP     CBEGIN
FD03 C335FD             JMP     CSWAP
```

Listing Five (Continued)

```
                ;
                ;SET PARAMETERS, ALLOCATE STACKS, AND SET ENTRIES
                ;
0008 =          STAKSIZ EQU     8               ; SET STACK SIZE
0000 =          GO      EQU     0               ; SET STARTING COROUTINE
FD06                    DS      STAKSIZ*2-2
FD14 0001       TOSTRT  DW      100H            ; ENTRY TO FORTH
FD16                    DS      STAKSIZ*2-2
FD24 28F0       T2STRT  DW      0F028H          ; ENTRY TO NORTHSTAR DOS
                ;STACKTAB IS START OF ARRAY FOR STORAGE
                ;       OF STACK POINTERS
                ;CURSTACK POINTS TO CURRENT POINTER
                ;       IN STACKTAB
FD26 14FD       STAKTAB DW      TOSTRT
FD28 24FD               DW      T2STRT
                ;
FD2A 26FD       CURSTAK DW      STAKTAB+GO
                ;
FD2C 2A2AFD     CBEGIN  LHLD    CURSTAK
FD2F 7E                 MOV     A,M
FD30 23                 INX     H
FD31 66                 MOV     H,M
FD32 6F                 MOV     L,A
FD33 F9                 SPHL
FD34 C9                 RET
FD35 210000     CSWAP   LXI     H,0
FD38 39                 DAD     SP
FD39 EB                 XCHG
FD3A 2A2AFD             LHLD    CURSTAK
FD3D 73                 MOV     M,E
FD3E 23                 INX     H
FD3F 72                 MOV     M,D
FD40 CD4CFD             CALL    GETADDR
FD43 222AFD             SHLD    CURSTAK
FD46 5E                 MOV     E,M
FD47 23                 INX     H
FD48 56                 MOV     D,M
FD49 EB                 XCHG
FD4A F9                 SPHL
FD4B C9                 RET
FD4C 2126FD     GETADDR LXI     H,STAKTAB
FD4F 85                 ADD     L
FD50 6F                 MOV     L,A
FD51 D0                 RNC
FD52 24                 INR     H
FD53 C9                 RET
```

End Listings

Linking CP/M Functions to Your High-Level Program

Abstract

There are various useful operations generally performed by the operating system of a microcomputer – directory listing, file deletion, and other file manipulations. Bringing these within the scope of a high-level language greatly expands a programmer's capabilities. The system functions included in CP/M* allow short assembly language programs to be linked to programs written in higher-level languages, permitting file manipulation during program execution. The program listed here, DEL, can be used to delete files during the execution of a Fortran program, and is based on the two CP/M functions, Search for First and Delete File.

All CP/M function calls are made in the same general manner. A value identifying the function desired is loaded into register C. An address, if necessary, may be loaded into DE, identifying the location of required information (for instance, the name of a file to be compared to directory listings). The command, CALL BDOS, then results in execution of the function.

DEL, the assembly-language program shown in Listing One, can be used to delete files during the execution of a Fortran program. It uses two of these functions, Search for First and Delete File. The short Fortran program, shown in Listing Two, illustrates the method of calling an assembly-language subroutine. CALL DEL(NAME,NUM) calls the subroutine DEL and places the address of NAME (the filename) in HL and the address of NUM (the output parameter which indicates what has happened to the file) in DE. During execution of DEL, placing the desired output at the location initially in DE then ensures that the variable NUM will contain the desired output upon return to Fortran.

The first segment of DEL, lines 7-15, saves the address of the output parameter and sets up a working file control block (FCB). The subroutine RO then calls the function Search for First to determine whether the file specified by the user is actually present. If not, it returns; if present, it examines the ninth byte of the filename in the directory to determine if it is read-only (R/O). A nonexistent or R/O file results in a specific value being returned in the accumulator: FFH if the file does not exist, and 1 if it is R/O. If the file is present and read/write (R/W), the subroutine DELT is called which uses the CP/M function Delete File to delete the file. It then returns 0 in the accumulator. Since each of the three cases – nonexistent, R/O, and R/W – results in a unique accumulator value, a value can be placed at the output parameter address to identify what has occurred when control is returned to the high-level language calling program. A value of -1 indicates a nonexistent file; 0, a deleted file, and 1, a R/O file. In fact, this program has an added advantage over the system ERA command since an attempt to delete a R/O file does not cause the system to suspend operation.

It should be noted that the CP/M Delete function will, in case of an ambiguous filename, delete all files that fit the pattern given. However, since the program DEL is designed to check only one filename to ensure that it is R/W, an attempt to delete several files, only some of which are marked R/O, may generate the usual error message and suspend system operation. A more extensive program, using both the Search for First and the Search for Next functions to check all filenames that fit an ambiguous description, could circumvent this problem.

These particular programs were written using an Apple II computer with a Z-80 Softcard. The CP/M operating system was used, along with the Microsoft version of Fortran and assembly language designed for the M80 assembler provided in Microsoft's ALDS.

The program DEL demonstrates only one of many possible combinations of CP/M system functions. A large number of file manipulations can be performed within the context of higher-level languages when these functions are used. Considering the space limitations of microcomputers, their potential usefulness in any program which involves the creation or alteration of large data files is considerable.

by Wendy Weiger

Wendy A. Weiger, 2816 Blazer Court, Wheaton, MD 20906.

**CP/M is a trademark of Digital Research, Inc.*

Editor's Note: Wendy Weiger was a 1982 graduate of Harvard in chemistry. She spent the summer working at Lamont-Doherty Geological Observatory of Columbia University on a geochemistry research project. While there, she developed the program discussed in the above article to be used in conjunction with a statistical analysis package. It illustrates a feature of CP/M that would be valuable to many of its users.

DDJ

Listing One — DEL

```
 1:          ENTRY   DEL             ;ON ENTRY,HL HAS THE FILENAME ADDRESS,DE
 2:                                  ;THE OUTPUT PARAMETER ADDRESS
 3: BDOS     EQU     05H             ;THESE 3 ADDRESSES ARE DEFAULT VALUES
 4: BUFF     EQU     81H
 5: FCB      EQU     05CH
 6:
 7: DEL:     LD      (TEMP),DE       ;SAVE ADDRESS OF 2ND (OUTPUT) PARAMETER
 8:          LD      DE,FCB+1        ;WANT TO MOVE FILENAME TO FCB+1
 9:          LD      BC,0BH          ;HL HAS FILENAME ADDRESS,1ST INPUT PARAM
10:          LDIR                    ;FILENAME TO FCB BYTES 1-11
11:          LD      A,0H
12:          LD      (FCB),A
13:          LD      (FCB+12),A
14:          LD      (FCB+15),A
15:          LD      (FCB+32),A      ;COMPLETES WORKING FCB
16:          CALL    RO              ;CHECKS IF FILE PRESENT AND R/W
17:          CP      0FFH            ;WILL BE 0FFH IF FILE NOT FOUND
18:          JP      Z,LAST
19:          CP      01H             ;WILL BE 1 IF FILE R/O
20:          CALL    NZ,DELT         ;DELETES FILE IF PRESENT AND R/W
21: LAST:    LD      D,A
22:          LD      E,0H            ;VALUE IN A MADE INTO 2-BYTE QUANTITY
23:          CP      0FFH            ;0FFH IF FILE NOT FOUND
24:          JP      NZ,NEXT
25:          LD      E,0FFH          ;OUTPUT WILL BE -1 IF FILE NOT FOUND
26: NEXT:    LD      HL,(TEMP)       ;TEMP HAS ADDRESS OF OUTPUT PARAMETER
27:          LD      (HL),D
28:          INC     HL
29:          LD      (HL),E
30:          RET                     ;ALL DONE
31:
32: RO:      LD      C,11H           ;TO CHECK IF FILE PRESENT AND R/W
33:          LD      DE,FCB          ;ENTER CALLING PARAMS FOR SEARCH FOR 1ST
34:          CALL    BDOS
35:          CP      0FFH            ;0FFH IN A IF FILE NOT FOUND
36:          RET     Z               ;RETURNS IF FILE NOT FOUND.A IS FFH
37:          ADD     A,A
38:          ADD     A,A
39:          ADD     A,A
40:          ADD     A,A
41:          ADD     A,A
42:          LD      C,A
43:          LD      B,00H
44:          LD      HL,BUFF         ;BUFF+(2**5)*A GIVES LOCATION OF
45:          ADD     HL,BC           ;     FILENAME TAKEN FROM DIRECTORY
46:          LD      BC,08H
47:          ADD     HL,BC           ;ADDRESS OF 9TH BYTE OF FILENAME
48:          LD      A,80H           ;80H IS 10000000B
49:          AND     (HL)            ;RESULT 1 IF AND ONLY IF FILE R/O
50:          RET     Z               ;RETURNS IF FILE FOUND AND R/W
51:          LD      A,01H           ;A SET TO 1 IF FILE R/O
52: FIN:     RET
53: DELT:    LD      C,13H           ;WILL DELETE FILE
54:          LD      DE,FCB          ;ENTER CALLING PARAMS FOR DEL FUNCTION
55:          CALL    BDOS
56:          LD      A,0H            ;A SET TO 0 IF FILE DELETED
57:          RET
58: TEMP:    DS      2
59:          END
60:
```

Listing Two — Calling the Assembly-language Routine

```
 1:          BYTE NAME(11)
 2: 100      FORMAT(11A1)
 3: 200      FORMAT(' ENTER FILENAME: 8-CHAR NAME,3-CHAR EXTENSION--FILL'/
 4:     U    ' WITH BLANKS IF NECESSARY TO EXTEND NAME TO 8 CHARS (11A1)'/)
 5: 300      FORMAT(' FILE DELETED'/)
 6: 400      FORMAT(' FILE NOT FOUND'/)
 7: 500      FORMAT(' FILE R/O'/)
 8:          WRITE(3,200)
 9:          READ(3,100)NAME
10:          CALL DEL(NAME,NUM)
11:          IF(NUM.EQ.-1)WRITE(3,400)
12:          IF(NUM.EQ.1)WRITE(3,500)
13:          IF(NUM.EQ.0)WRITE(3,300)
14:          STOP
15:          END
```

End Listing

CP/M EXCHANGE

by Gene Head

Last month I mentioned a hot DIR.ASM routine. Well, here it is. DIR.ASM assembles to become a direct replacement for the DIR command in the CP/M CCP. DIR.COM will display a sorted and sized directory along with space and directory entries remaining on the entire disk. DIR.ASM evaluates hidden (SYS) files as well as files in all user areas when computing space and directory entries remaining unused.

CP/M Function #31 is the heart of the disk sizing code and should make this program transportable to any CP/M system without modification! I have *not* tested this on a hard-disk system but it seems to fly OK on different 8" and 5" floppy machines.

Bob Blum of Norcross, Georgia has promised a follow-up contribution detailing Function #31 and all data contained in the Disk Parameter Block.

Remember, this is your column. Make a contribution!

DDJ

DIR.ASM Listing

```
;
;   FROM: HEAD QUARTERS - CORVALLIS, OREGON  97330
;   FOR:  THE CP/M EXCHANGE
;   DATE: 10-31-82   (FOR JANUARY '82 ISSUE)
;
;        DIR.ASM
;
;   THIS PROGRAM WILL REPLACE THE DIR COMMAND
;   IN THE CP/M CCP.
;
;   YOU CAN NAME THIS PROGRAM DIR.COM AND DISABLE
;   THE DIR COMMAND IN THE CCP BY DOING A 'GET' SYSTEM
;   SKIP THE 'PUT' SYSTEM BUT DO A SAVE xx CPM.COM
;   NOW DDT CPM.COM AND DUMP MEMORY UNTIL YOU SEE
;   THE ASCII COMMANDS DIR, ERA, TYPE, REN.  CHANGE
;   DIR TO 'DIRx' THEN 'PUT' THE MODIFIED SYSTEM.
;
;   THE SORTED DIRECTORY WILL BE DISPLAYED WITH
;   FILE SIZES IN KILO-BYTES AND AMBIGUOUS FILE
;   REFERENCES ARE ACCEPTABLE (*.* ???????.???)
;
;   THE FOOTER BANNER WILL LOOK LIKE:
;
;   DRIVE: xn    FILES:  a (b)       BYTES:  ck  (dk)
;
;   Where x=drive, n=user
;         a=number of files found
;         b=number of directory entries available
;        ck=kilo-bytes used in found files
;        dk=kilo-bytes remaining unused on the disk
;
;   R/O FILES ARE DISPLAYED USING LOWER-CASE LETTERS
;
;   OPTIONALLY, SPECIFIED USER AREAS CAN BE DECLARED
;   BY PLACING ONE OR TWO DIGITS IN THE SECOND FCB.
;   TO SEE ALL THE .COM FILES ON DRIVE B: USER AREA 4 TYPE:
;   DIR B:*.COM 4   <or>      DIR B:*.COM 04
;   UP TO 15 USER AREAS ARE SUPPORTED IN THIS VERSION.
; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ;
```

```
        ORG     100H

;DEFINE THE EQUATES


;SYSTEM FUNCTIONS
FCONOUT EQU     2       ;CONSOLE OUTPUT
FSTRING EQU     9       ;PRINT STRING
FVERS   EQU     12      ;RETURN CP/M VERSION
FRESET  EQU     13      ;RESET DISK SYSTEM
FSELDSK EQU     14      ;SELECT DISK
FSEARCH EQU     17      ;SEARCH DIRECTORY
FNEXT   EQU     18      ;SEARCH NEXT DIRECTORY
FGETDSK EQU     25      ;RETURN CURRENT DISK DRIVE
FDPB    EQU     31      ;GET DPB ADDRESS
FUSER   EQU     32      ;GET USER NUMBER
FSIZE   EQU     35      ;GET FILE SIZE


;OTHER STUFF
BDOS    EQU     5       ;BDOS ENTRY POINT


FCB     EQU     5CH     ;FILE PCONTROL BLOCK
FCB2    EQU     6CH     ;SECOND FCB
BUFF    EQU     80H     ;DIRECTORY BUFFER
WRITE   EQU     10      ;R/O CODE OFFSET
RECNT   EQU     15      ;RECORD COUNT OFF-SET
SYSTEM  EQU     9       ;SYSTEM BYTE OFF-SET
MAXLINE EQU     3       ;DISPLAYED ENTRIES PER LINE
CR      EQU     13      ;CARRIAGE RETURN
LF      EQU     10      ;LINE FEED


;BEGIN PROGRAM CODE
START   LXI     H,0
        DAD     SP
        SHLD    STACK   ;SAVE ENTRY STACK
        LXI     SP,STACK

        MVI     C,FRESET
        CALL    BDOS


;CHECK FOR PROPER VERSION OF CP/M
        MVI     C,FVERS
        CALL    BDOS
        CPI     20
        LXI     D,BADVER        ;BAD VERSION
        JC      EXIT            ;EXIT IF NOT CP/M 2


;INITIALIZE THE DISK, THE USER, AND THE FCB
        CALL    INITIALIZE
```

(Continued on next page)

DIR.ASM Listing (Listing continued, text begins on page 34)

```
;DETERMINE SPACE REMAINING ON DISK
;AND REMAINING DIRECTORY ENTRIES
        CALL    CAPACITY        ;COMPUTE DISK CAPACITY

;READ SELECTED ENTRIES FROM DIRECTORY INTO BUFFER
        CALL    RESET           ;RESET THE BUFFER POINTER
        CALL    READ            ;READ IN SELECTED DIRECTORY

;SORT SELECTED INTRIES IN THE BUFFER
        CALL    RESET           ;RESET BUFFER POINTER
        CALL    SORT            ;SORT THE DIRECTORY

;DISPLAY SORTED, SELECTED ENTRIES FROM THE BUFFER
        CALL    RESET           ;RESET THE BUFFER POINTER
        CALL    DISPLAY         ;DISPLAY SORTED DIRECTORY

;DISPLAY DISK CAPACITY AND FILES/SPACE USED
        LXI     D,FOOTER        ;POINT TO FOOTER BANNER

;DISPLAY MESSAGE POINTED TO BY DE THEN EXIT
EXIT    MVI     C,FSTRING
        CALL    BDOS
        LDA     USERC
        MOV     E,A
        MVI     C,FUSER
        CALL    BDOS
        LHLD    STACK
        SPHL                    ;RESTORE ENTRY STACK
        RET                     ;...THEN EXIT


 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*               SUBOUTINES FOLLOW                                 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

;INITIALIZE DISK, USER, AND FCB
;-------->
INITIALIZE      EQU     $

;GET TO THE PROPER DRIVE IF NOT DEFAULT DRIVE
        LDA     FCB
        ORA     A
        PUSH    PSW
        MOV     E,A
        DCR     E
        POP     PSW
        MVI     C,FSELDSK
        CNZ     BDOS
```

```
;NOW GET THE SELECTED DISK (IT MIGHT BE
;THE DEFAULT DRIVE) AND SAVE IT AT 'DRIVE'
        MVI     C,FGETDSK
        CALL    BDOS
        STA     DISK
        ADI     'A'             ;MAKE IT ASCII
        STA     DRIVE           ;SAVE IT IN FOOTER

;NOW GET THE USER NUMBER, CONVERT IT TO DECIMAL,
;SAVE IT AT 'USER'
        MVI     C,FUSER
        MVI     E,0FFH
        CALL    BDOS
        STA     USERC           ;SAVE CURRENT USER
        LHLD    FCB2+1          ;GET ASSIGNED USER
        MOV     A,L
        CPI     ' '
        JNZ     PASTY
        MOV     A,H
        CPI     ' '
        JZ      PASTX           ;NO NEW USER GIVEN

PASTY   CALL    DEC2BIN         ;CONVERT ASCII TO BINARY
        STA     USERX           ;SAVE ASSIGNED USER
        MOV     E,A
        MVI     C,FUSER
        CALL    BDOS            ;SET NEW USER

PASTX   LDA     USERX
        MOV     L,A
        MVI     H,0
        LXI     D,USER          ;SAVE ASCII DECIMAL
        CALL    XXDECIMAL       ;IN FOOTER BANNER

;FILL IN THE FCB WITH '?'s IF IT IS EMPTY
        LDA     FCB+1
        CPI     ' '
        LXI     D,FCB+1
        LXI     H,DUMMY         ;DUMMY FCB IS '????????.???'
        MVI     B,11
        CZ      MOVH2D

        RET

;CONVERT ASCII CHARACTERS IN HL TO BINARY VALUE IN A
;-------->
DEC2BIN EQU     $
        MOV     A,H
        CPI     ' '             ;IS THIS SINGLE DIGIT?
        JNZ     DO1
        MOV     H,L             ;CONVERT SINGLE DIGIT TO
        MVI     L,0             ;DOUBLE DIGIT LIKE 1=01
DO1     MOV     A,L
        SUI     '0'             ;MAKE BINARY
```

(Continued on next page)

DIR.ASM Listing (Listing continued, text begins on page 34)

```
          ANI     1
          MVI     B,0
          JZ      DO2
          MVI     B,10            ;ADD 10 IF > 9
DO2       MOV     A,H
          SUI     '0'             ;MAKE BINARY
          ADD     B
          CPI     16              ;TEST FOR MAX OF 15
          RC                      ;RETURN IF < 16 ELSE
          LDA     USERC           ;...DEFAULT TO CURRENT USER
          RET

;RESET THE DIRECTORY BUFFER POINTER
;THE TO TOP OF THE DIRECTORY BUFFER
;-------->
RESET     EQU     $
          LXI     H,BUFFER
          SHLD    DIRBUFP
          RET

;DISPLAY THE BUFFER (MAXLINE) WIDE
;-------->
DISPLAY   EQU     $
          LDA     SFILES
          ORA     A
          JNZ     DISP1           ;TEST FOR EMPTY BUFFER
          LXI     D,NOENT
          MVI     C,FSTRING
          CALL    BDOS
          RET

DISP1     LXI     H,XXNUL
          MVI     M,' '           ;DISPLAY LEADING ZEROS
          LXI     H,0             ;...AS BLANK SPACES (JUSTIFY)
          SHLD    TSIZE
          SHLD    TFILES



          CALL    NEWLINE
D1        PUSH    B
          LHLD    DIRBUFP
          PUSH    H               ;SAVE CURRENT POINTER
          LDA     XXRECL
          MVI     D,0
          MOV     E,A
          DAD     D               ;GET NEXT RECORD POINTER
          SHLD    DIRBUFP         ;SAVE NEXT ENTRY POINT


          DCX     H               ;GET TO CURRENT RECORD SIZE

;DISPLAY R/O FILES AS LOWER CASE ASCII
          DCX H!  DCX  H!  DCX  H ;GET TO THE R/O BYTE
          MOV     A,M             ;...AND SAVE IT IN 'A'
```

```
        RAR!    RAR
        ANI     20H             ;SET LOWER CASE MASK
        STA     MASK            ;...IF READ/ONLY FILE
        INX H!  INX  H!  INX  H ;GO BACK TO RECORD COUNT

;CHECK FILE SIZE FOR ZERO
        MOV     L,M             ;SIZE IN HL
        MVI     H,0
        MOV     A,L
        ORA     A

;IF SIZE IS ZERO THIS IS EITHER A NULL FILE OR A
;FILE WITH A RECORD COUNT OF 80H.
;IN EITHER CASE GO USE THE GETSIZE
;SUBROUTINE TO PROPERLY SIZE THE FILE.

        CZ      GETSIZE

        XCHG
        LHLD    TSIZE           ;SUM TOTAL SIZE
        DAD     D
        SHLD    TSIZE
        LHLD    TFILES
        INX     H
        SHLD    TFILES          ;BUMP FILE COUNTER
        XCHG


        LXI     D,KSIZE         ;SAVE FILE SIZE
        CALL    XXDECIMAL       ;...IN K-BYTES
        POP     H               ;RESTORE DIRECTORY POINTER
        MVI     C,8
        CALL    NPRINT          ;FILE NAME
        MVI     E,' '
        CALL    PRINT           ;...SPACE
        MVI     C,3
        CALL    NPRINT          ;......TYPE
        XRA     A
        STA     MASK            ;CLEAR LOWER-CASE MASK
        LXI     H,KSIZE+1
        MVI     C,5
        CALL    NPRINT          ;.........SIZE


        POP     B               ;GET WIDTH COUNTER
        DCR     B
        JNZ     D2
```

(Continued on next page)

DIR.ASM Listing (Listing continued, text begins on page 34)

```
;GET HERE ONLY AFTER LAST ENTRY PER LINE
        CALL    NEWLINE
        CALL    BUMP
        JNZ     D1
        JMP     BAN1

;GET HERE AFTER ANY ENTRY OTHER THAN LAST PER LINE
D2      LXI     H,OLDL
        MOV     C,M
        INX     H
        CALL    NPRINT

;GET HERE AFTER EACH DISPLAYED ENTRY
D3      CALL    BUMP
        JNZ     D1
        CALL    NEWLINE
        JMP     BAN1

BUMP    LDA     SFILES          ;COUNT DOWN ENTRY COUNTER
        DCR     A
        STA     SFILES
        RET

;START A NEW DISPLAY LINE AND INITIALIZE WIDTH COUNTER
NEWLINE MVI     E,CR
        CALL    PRINT
        MVI     E,LF
        CALL    PRINT
        MVI     B,MAXLINE
        RET


;------->
CAPACITY        EQU     $

;USING THE CP/M FUNCTION TO ACCESS THE DISK PARAMETER BLOCK,
;FIND MAX DISK SPACE, MAX DIRECTORY ENTRIES
;SEE PAGE 27 OF CP/M ALTERATION GUIDE (C) DIGITAL RESEARCH 1979

;       BLS             BSH             BLM
;       ------------------------------------
;       1024             3               7
;       2048             4              15
;       4096             5              31
;       8192             6              63
;       16384            7             127


        MVI     C,FDPB          ;RETURN DISK PARAMETER ADDRESS
        CALL    BDOS
        INX     H
        INX     H
        MOV     A,M             ;GET SHIFT FACTOR BSH
        DCR     A
```

```
        DCR     A
        STA     BSH

        INX     H
        MOV     A,M             ;GET BLM
        STA     BLM

        INX     H               ;SKIP EXM BYTE

        PUSH    H               ;SAVE DPB POINTER
        LXI     H,1024/2
        LDA     BSH             ;GET SHIFT FACTOR
SIZE1   DAD     H
        DCR     A
        JNZ     SIZE1
        SHLD    BLS             ;BLS = 512 * ( (BSH-2) * 2 )
        POP     H               ;RESTORE DPB POINTER

        INX     H
        MOV     E,M
        INX     H
        MOV     D,M             ;GET DOUBLE BYTE DSM
        INX     D
        INX     H

        XCHG
        SHLD    DISKSIZ         ;SAVE DSM
        XCHG

        MOV     E,M
        INX     H
        MOV     D,M             ;GET DOUBLE BYTE DRM
        XCHG
        INX     H               ;MAXENT = DRM + 1
        SHLD    MAXENT          ;SAVE MAX DIR ENTRIES

        DAD     H               ;COMPUTE TOTAL SPACE
        DAD     H               ;..REQUIRED FOR
        DAD     H               ;....THE DIRECTORY
        DAD     H               ;32 BYTES PER ENTRY
        DAD     H               ;OR (DRM+1)*32

        XCHG
        LHLD    BLS
        DCX     H
        DAD     D
        LDA     BSH             ;COMPUTE NUMBER OF
        MOV     C,A             ;..BLOCKS THAT HAVE
        MOV     A,H             ;....BEEN RESERVED FOR
        RAR                     ;......THE DIRECTORY
        ANI     7EH
SIZE2   RAR
        DCR     C
        JNZ     SIZE2
```

DIR.ASM Listing (Listing continued, text begins on page 34)

```
          CMA
          MOV     C,A
          MVI     B,0FFH
          INX     B
          LHLD    DISKSIZ          ;SUBTRACT DIRECTORY SPACE
          DAD     B
          LDA     BSH
SIZE3     SHLD    DISKSIZ          ;SAVE TOTAL REMAINING SPACE
          DCR     A
          JZ      READALL
          DAD     H
          JMP     SIZE3

SIZE4     SHLD    DISKSIZ          ;SAVE TOTAL SPACE FOR FILES


READALL   MVI     C,FSEARCH

LOOP1     LXI     D,DUMMY
          MVI     A,'?'            ;SEARCH EVERY DIRECTORY ENTRY
          STAX    D
          CALL    BDOS
          ORA     A                ;END OF DIRECTORY?
          JM      BANNER           ;GO FILL THE BANNER
          CALL    SIZFIL
          JZ      LOOP2
          LHLD    TFILES
          INX     H                ;COUNT EVERY ENTRY
          SHLD    TFILES
          LHLD    TSIZE
          DAD     B                ;ADD UP TOTAL SPACE USED
          SHLD    TSIZE
LOOP2     MVI     C,FNEXT
          JMP     LOOP1
;FILL BANNER WITH DISK CAPACITY INFO
;-------->
BANNER    EQU     $
          XRA     A
          STA     XXNUL
          LHLD    TFILES
          XCHG
          LHLD    MAXENT
          CALL    XXDMINUS
          DAD     D
          LXI     D,FREEF
          CALL    XXDECIMAL        ;SHOW FREE FILES
          LHLD    TSIZE
          XCHG
          LHLD    DISKSIZ
          CALL    XXDMINUS
          DAD     D
          LXI     D,FREEB
          CALL    XXDECIMAL
          RET
```

```
;FILL BANNER WITH SELECTED FILE INFO
;-------->
BAN1    EQU     $
        XRA     A
        STA     XXNUL
        LHLD    TFILES
        LXI     D,FILES
        CALL    XXDECIMAL
        LHLD    TSIZE
        LXI     D,BYTES
        CALL    XXDECIMAL
        RET

;GET THE SIZE OF THE FOUND FILE
;-------->
SIZFIL: EQU     $

;CALLED IMMEDIATELY AFTER A SUCCESSFUL SEARCH
;OR SEARCH NEXT FUNCTION CALL.
;ON EXIT:
;HL POINTS TO THE FILE IN THE DIRECTORY.
;BC HOLDS THE KILO-BYTE FILE SIZE.

;ZERO SET = SKIP ALL DELETED FILES.
;CARRY SET = FULL EXTENT (COUNT ENTRY BUT DON'T SIZE)
;ZERO & CARRY CLEAR = COUNT THE ENTRY AND SIZE IT


        LXI     B,0             ;ASSUME ZERO FILE LENGTH
        LXI     H,BUFF          ;DIRECTORY BUFFER
        ANI     03              ;0-3
        RRC!  RRC!  RRC         ;A=A*32
        ADD     L
        MOV     L,A
        PUSH    H               ;HL POINTS TO FOUND FILE
        MOV     A,M
        ORA     A               ;CLEAR THE CARRY FLAG
        CPI     0E5H            ;CHECK THE ZERO FLAG
        JZ      SKIPALL         ;SKIP DELETED ENTRIES
        LXI     D,RECNT         ;RECORD COUNT FIELD
        DAD     D
        MVI     A,7FH
        ANA     M
        JZ      SKIPSIZ         ;SKIP SIZE OF EXTENDED FILES
        MOV     B,M
        LDA     BLM
        CMA
        MOV     C,A
        CMA
        ADD     B
        ANA     C
        RRC!  RRC!  RRC
        ANI     1FH             ;K-BYTES IN RECORD COUNT
        MOV     C,A             ;...OF LAST EXTENT
```

(Continued on next page)

DIR.ASM Listing (Listing continued, text begins on page 34)

```
          DCX     H
          DCX     H                 ;BACK UP TO EXTENT BYTE
          DCX     H
          MOV     A,M               ;GET NUMBER OF EXTENTS
          RLC!  RLC!  RLC!  RLC     ;
          ANI     0F0H
          ADD     C
          MOV     M,A               ;SIZE IN K-BYTES IN EX FIELD
          MOV     C,A
          MVI     B,0               ;FILE SIZE IN BC
          XRA     A                 ;CLEAR CARRY
SKIPSIZ   INR     A                 ;NON-ZERO
SKIPALL   POP     H                 ;HL POINTS TO DIRECTORY ENTRY
          RET

;USE CP/M FUNCTION #35 TO GET THE ACTUAL FILE SIZE
;-------->
GETSIZE   EQU     $
;ON ENTRY HL POINTS TO A FILE IN THE BUFFER
;ON EXIT HL=FILE SIZE IN K-BYTES
          POP     B! POP D          ;GET/SAVE START OF BUFFER
          PUSH    D! PUSH  B        ;...ENTRY IN DE
          LXI     H,DUMMY
          XCHG
          PUSH    H!  PUSH  D
          INX     D                 ;PASS UP DRIVE FIELD
          MVI     B,12
          CALL    MOVH2D            ;FILL DUMMY WITH FILE TO SIZE
          POP     D!  POP  H
          PUSH    D!  PUSH  H
          MVI     C,FSIZE           ;SIZE THE FILE
          CALL    BDOS
          POP     H
          POP     D
          XCHG
          LHLD    DUMMY+33          ;MAKE SIZE = K-BYTES
          MOV     A,L
          ANI     7
          PUSH    PSW
          MOV     A,L
          RAR!    RAR!    RAR
          ANI     1FH
          MOV     L,A
          MOV     A,H
          RAL!    RAL!    RAL!    RAL!    RAL
          ANI     0E0H
          ORA     L
          MOV     L,A
          MOV     A,H
          RAR!    RAR!    RAR
          ANI     1FH
          MOV     H,A
          POP     PSW
          RZ
          INX     H                 ;ADD ONE IF OVER-FLOWED
          RET
```

```
;READ EACH DIRECTORY ENTRY INTO THE DIRECTORY BUFFER
;-------->
READ    EQU     $

;HL POINTS TO THE DIRECTORY ENTRY OF 12 CHARACTERS
;1-8 FILE NAME, 9-11 FILE TYPE, 12 FILE SIZE IN K-BYTES

        LXI     H,0
        SHLD    TFILES          ;CLEAR FILE COUNT
        SHLD    TSIZE           ;CLEAR TOTAL SIZE
        MVI     C,FSEARCH
READ1   LXI     D,FCB
        CALL    BDOS
        ORA     A               ;EXIT IF NO FILES
        RM

        CALL    SIZFIL
        JZ      READ2           ;SKIP DELETED FILES
        JC      READ2           ;SKIP FULL EXTENTS
        LDA     USERX
        CMP     M
        JNZ     READ2           ;SKIP NON-CURRENT USER
        INX     H               ;PAST DR FIELD
        PUSH    H
        LXI     D,SYSTEM        ;LOOK AT SYSTEM BYTE
        DAD     D
        MOV     A,M
        RLC
        POP     H
        JC      READ2           ;SKIP SYSTEM FILES
        PUSH    H
        LHLD    DIRBUFP         ;GET CURRENT OUTPUT
        XCHG                    ;POINTER IN DE
        LXI     H,12            ;BUMP TO NEXT ENTRY
        DAD     D
        SHLD    DIRBUFP         ;SAVE NEXT POSITION
        LXI     H,SFILES
        INR     M               ;COUNT FILES IN BUFFER
        POP     H               ;HL=SOURCE  DE=DESTINATION
        MVI     B,12
        CALL    MOVH2D

READ2   MVI     C,FNEXT
        JMP     READ1



;DECIMAL CONVERSION
;-------->
XXDECIMAL:      EQU     $

; FROM DAVE FRIEDMAN'S SLOT MACHINE PROGRAM
; DDJ #50 NOV/DEC 1980
```

(Continued on next page)

DIR.ASM Listing (Listing continued, text begins on page 34)

```
;THIS SUBROUTINE CONVERTS  THE VALUE IN (HL)
;TO DECIMAL ASCII AND  STORES THE CHARACTERS
;LEFT JUSTIFIED (NO LEADING SPACES OR ZEROS)
;TO THE CONSECUTIVE MEMORY LOCATIONS POINTED
;TO BY (DE).

;ALL REGISTERS ARE PRESERVED

;ALL LABELS ARE 'XXLABEL' TO AVOID COFLICT
;IN EXISTING LABELS.
        PUSH H!  PUSH  D!  PUSH  B!  PUSH  PSW

        MVI     B,0             ;SUPRESS LEADING ZEROS
        PUSH    D
        LXI     D,-10000
        CALL    XXDIV
        POP     D!  STAX  D!  INX  D!  PUSH  D
        LXI     D,-1000
        CALL    XXDIV
        POP     D!  STAX  D!  INX  D!  PUSH  D
        LXI     D,-100
        CALL    XXDIV
        POP     D!  STAX  D!  INX  D!  PUSH  D
        LXI     D,-10
        CALL    XXDIV
        POP     D!  STAX  D!  INX  D!  PUSH  D
        LXI     D,-1
        CALL    XXDIV
        POP     D
        LDA     XXNUL
        CMP     C
        MOV     A,C
        JNZ     XXLASTD


        MVI     A,'0'   ;SHOW AT LEAST ONE ZERO
XXLASTD STAX    D
        POP  PSW!  POP  B!  POP  D!  POP  H
        RET

XXDIV   MVI     C,'0'-1
XXDIV1  INR     C
        DAD     D
        JC      XXDIV1
        CALL    XXDMINUS
        DAD     D
        MOV     A,C
        ORA     B
        MOV     B,A
        CPI     '0'
        JNZ     XXSURP
        LDA     XXNUL
        MOV     C,A
XXSURP  MOV     A,C
        RET
```

```
;MAKE DE=DE*-1 FOR DOUBLE SUBTRACTION
XXDMINUS        EQU     $
        MOV     A,E!    CMA!    MOV     E,A
        MOV     A,D!    CMA!    MOV     D,A
        INX     D
        RET


;SORT THE BUFFER
;------->
SORT    EQU     $

;  FROM INTERFACE AGE NOV-78 - CHRIS TERRY

XXSORT  EQU     $
        LDA     XXRECL
        MVI     D,0
        MOV     E,A
        CALL    XXDMINUS
        LHLD    DIRBUFP
        DAD     D
        SHLD    DIRBUFP
        LDA     SFILES
        MOV     H,A
XXNEWPS MOV     A,H     ;START NEW PASS
        ORA     A       ;CLEAR CARRY
        RAR             ;CUT IN HALF
        ORA     A       ;SE STATUS FLAGS
        MOV     H,A     ;SAVE NEW VALUE
        RZ              ;RETURN WHEN DONE
        LDA     SFILES
        SUB     H
        MOV     D,A
        MVI     C,1
XXSORT1 MOV     B,C
XXSORT2 MOV     A,B
        ADD     H
        MOV     E,A

;SET I & L ADDRESS POINTERS
XXDISPI PUSH    H!  PUSH  D!  PUSH  B
        MOV     A,B
        CALL    XXSETAD
        SHLD    XXCRCEL
        POP     B!  POP   D!  POP   H

XXDISPL PUSH    H!  PUSH  D!  PUSH  B!
        MOV     A,E
        CALL    XXSETAD
        SHLD    XXCRCL1
        POP     B!  POP D!  POP   H

XXCOMP  PUSH    H!  PUSH  D!  PUSH  B
        LHLD    XXCRCEL
```

(Continued on next page)

DIR.ASM Listing (Listing continued, text begins on page 34)

```
          XCHG
          LHLD     XXCRCL1
          LDA      XXRECL
          MOV      B,A
          CALL     XXSCOMP
          POP      B!  POP  D!  POP   H

XXTEST    ORA      A                ;SET THE CPU FLAGS
          JZ       XXTEST1          ;NO NEED TO SWAP ON ZERO
          JM       XXTEST1          ;...OR MINUS.
          CALL     XXSWAP
          MOV      A,B
          SUB      H
          MOV      B,A
          JZ       XXTEST1
          JC       XXTEST1
          JMP      XXSORT2
XXTEST1   INR      C
          MOV      A,D
          CMP      C
          JC       XXNEWPS
          JMP      XXSORT1

;COMPUTE RECORD ADDRESS
XXSETAD   MVI      D,0
          MOV      E,A
          LDA      XXRECL
          CALL     XXMULT
          XCHG
          LHLD     DIRBUFP
          DAD      D
          RET

;HL=E*A
XXMULT    LXI      H,0
          MVI      B,8
XXMULT1   DAD      H
          RLC
          JNC      XXNOT1
          DAD      D
XXNOT1    DCR      B
          JNZ      XXMULT1
          RET
XXSWAP    PUSH     H!  PUSH  D!  PUSH   B
          LDA      XXRECL           ;RECORD LENGTH
          MOV      B,A
          LHLD     XXCRCEL
          XCHG
          LHLD     XXCRCL1
XXSWAP1   LDAX     D
          MOV      C,M
          MOV      M,A
          MOV      A,C
          STAX     D
          INX      D
          INX      H
          DCR      B
```

```
        JNZ     XXSWAP1
        POP     B!  POP  D!  POP  H
        RET

;COMPARE ALL BITS 0-6, SKIP BIT 7 - MSB
XXSCOMP DCR     B
        JM      XXSCOM1
        MOV     A,M
        ANI     7FH
        MOV     C,A
        LDAX    D
        ANI     7FH
        SUB     C
        INX     D
        INX     H
        JZ      XXSCOMP
XXSCOM1 RET


;MOVE 'B' BYTES FROM (HL) --> (DE)
;------->
MOVH2D  MOV     A,M
        STAX    D
        INX     H
        INX     D
        DCR     B
        JNZ     MOVH2D
        RET


;PUT MESSAGE POINTED TO BY 'HL' ON THE SCREEN
;'C' IS LENGTH OF MESSAGE
;------->
NPRINT  EQU     $
        DCR     C
        RM
        MOV     E,M
        INX     H
        CALL    PRINT
        JMP     NPRINT

;DISPLAY THE CHARACTER PASSED IN 'E' AND MASK
;WITH BYTE IN 'MASK'.  (USED HERE TO FORCE LOWER
;CASE ON FLAGGED CHARACTERS)
;------->
PRINT   PUSH    H!  PUSH  D!  PUSH  B!  PUSH  PSW
        LDA     MASK
        ORA     E
        ANI     7FH
        MOV     E,A
        MVI     C,FCONOUT
        CALL    BDOS
        POP     PSW!  POP  B!  POP  D!  POP  H
        RET
```

(Continued on next page)

DIR.ASM Listing (Listing continued, text begins on page 34)

```
;STACK SPACE
        DS      20H
STACK   DW      0


;DATA STORAGE


;SORTING STORAGE
XXNUL   DB      0
XXCRCEL DW      0
XXCRCL1 DW      0
XXRECL  DB      12

;OTHER STORAGE
BLS     DW      0

DISK    DB      0       ;SAVE CURRENT DISK HERE

MASK    DB      0       ;UPPER/LOWER CASE MASK

USERX   DB      0       ;ASSIGNED USER NUMBER

USERC   DB      0       ;CURRENT USER NUMBER

DIRBUFP DW      0       ;DIRECTORY POINTER

BLM     DW      0       ;BLOCK LENGTH FROM FUNCTION #31

BSH     DB      0


DISKSIZ DW      0       ;MAX DISK SIZE

MAXENT  DW      0       ;MAX ENTRIES

TSIZE   DW      0       ;SIZE COUNTER

TFILES  DW      0       ;ENTRY COUNTER

SFILES  DW      0       ;FILES TO SORT

DUMMY   DB      '?','???????????',0,0,0,0,0
        DS      20

;MESSAGES
BADVER  DB      CR,LF,'REQUIRES CP/M 2',CR,LF,'$'

NOENT   DB      CR,LF,'NO FILE',CR,LF,'$'

;SUMMARY MESSAGE
FOOTER  DB      CR,LF
        DB      ' DRIVE '
```

```
DRIVE    DB       0
USER     DB       0,0,0,0,0
         DB       ': '
         DB       '   FILES: '
FILES    DB       0,0,0,0,'0 ('

FREEF    DB       '      )'

         DB       '  '
         DB       '   BYTES: '
BYTES    DB       0,0,0,0,'0k   ('
FREEB    DB       '      k) $'
         DB       CR,LF,'$'

OLDL     DB       OLDLEND-OLDL-1,' ',7CH,' '
OLDLEND  EQU      $

;TEMPROARY SIZE STORAGE
KSIZE    DB       0,0,0,0,0,'k '
BUFFER   EQU      $
```

End Listing

DR. DOBB'S CLINIC

by D. E. Cortesi

Diskette Senility?

Loren Amelang, of Philo, CA, presents two questions of considerable interest. Here they are.

"I seem to have a lot of trouble with 'bad sectors' on my diskettes. I can make a backup copy, verify it, leave it on the shelf for a month, and come back to find 'bad sector' errors. If I recopy a diskette that gives read errors, it will be fine for a month or two and then fail in a different spot. What happens to a first-quality Dysan or Maxell disk that makes it unreadable, yet leaves no permanent damage?

"Furthermore, when I get a read error, and can't edit or dump or otherwise access the file in question, I can *always* run the CRC check program (from the BDS C user group) and *it* can access the file in question! How can it be that one program can read a sector that another finds hardware-level (or at least BIOS-level) errors in?"

As to the first question, we once heard that the mylar of which the disk is made can pick up moisture from the air. That makes it swell and changes its radial dimensions, maybe just enough to throw tracks out of alignment. That might be one explanation for how a file could become unreadable; reformatting such a diskette would lay down new tracks that could be read. But we've never seen any warnings about this effect in print – maybe because there is no practical way for a typical user to control it, short of installing an air-conditioned computer room.

However, if that were the source of Amelang's problem it ought to show a seasonal variation, with failures most often at the start of California's rainy season (and again at the beginning of summer?). It's a pretty flaky theory, now that we see it in green and black. Has any reader another explanation? Or an answer to the second question – what kind of I/O is that BDS C User Group utility doing, and how can the rest of us do it?

Computer Catatonia?

Ernest Knipp, of Houston, TX, has seen cases where some unknown cause, presumably an action by an off-the-rails program, has hung his system up so high that he has had to power-down and -up again to get it going. (Knipp's system is not an IBM PC, which can be put in that state quite easily by any bug that clobbers either the 8086 interrupt vectors or MSDOS's chain of program-termination pointers.)

"The first time it happened," Knipp writes, "I thought something internal must be bent, broken, or burned out. It was quite a while before I caught on to the power-down-and-up maneuver. Your suggestion about the disk [*we proposed that the disk head might have been stepped back outside of track zero*] is interesting, but I don't think it applies; I have a front panel and what happens is that the computer never moves off the instruction at location zero, so it never has a chance to do anything with the disk. I have been guessing it was in some kind of wait state that is not reset by a front-panel reset."

Okay, readers – propose ways by which a (presumed program) bug could put a Z80 system into permanent catatonia at location zero, such that hitting the reset button on a front panel would not revive it.

Intern Truncated

Burks Smith, of Shawnee Mission, KS, objects to our comments about floating-point arithmetic in the October issue. Here is his rebuttal.

"You say that complaints that internal binary representation is more prone to truncation errors are 'misguided,' but go on to contradict yourself by pointing out two very good reasons not to use binary representation. They are that any truncation errors in decimal arithmetic are what people expect them to be with limited precision, and that some fractions that have a finite representation in base ten do not have a finite representation in base two. In fact, the decimal fraction 1/10 can not be represented exactly as a binary fraction, no matter how many bits are used, and this would seem to be reason enough to favor a BCD scheme for internal representation.

"Your admonition to never use floating-point arithmetic in 'commercial' calculations and suggestion to use fixed-point decimal representations available in PL/I and COBOL misses the point. We were talking about BASIC-80, and excuse me, but aren't high-level languages supposed to be for the convenience of people instead of CPUs? If something called 'double-precision arithmetic' can't take ten percent of a dollar and get *exactly* ten cents there shouldn't be any excuses. The answer is just plain wrong!"

Okay, what do we know? *Is* the answer just plain wrong? Or is the problem that people expect *mathematics* from a CPU that is only capable of *arithmetic* – and only approximate arithmetic at that? In what senses is a decimal base "better" than a binary one? Are there common ratios that have a finite representation in binary and not in decimal? The BCD-based float representation used in CBASIC has roughly the same precision as the double-precision binary representation used in MBASIC and BASCOM. Has anyone ever benchmarked the two methods for (a) commercial, or (b) scientific, computations? How much can we reasonably expect of computer arithmetic as done by current languages? What ought we to be demanding of the implementors of the next wave of compilers and interpreters? Is there a numerical analyst in the house?

Incidentally the choice of a radix for a float number is not limited to binary versus BCD. There are hardware systems that operate on octal and hexadecimal digit-groups, and we once read of a software package for a 16-bit machine that used a radix of 10,000 – each "digit" was a 16-bit number from 0 to 9,999.

Finally, if anyone wants to look into these matters in greater depth, here are a couple of sources. Those interested in the design of floating-point arithmetic methods might start with Knuth's *Seminumerical Algorithms*, especially section 4.2.2, "Accuracy of Floating-Point Arithmetic." For the user of a system, we understand that Richard Hamming's *Numerical Methods for Scientists and Engineers* is considered a classic.

Double your BASIC

Aubrey Hutchinson, of Pompano Beach, FL, has sent us a clutch of lovely sponses, which we plan to dole out sparingly. Here is the first. Were you aware that the built-in editor of MBASIC has the ability tc duplicate lines of code? It isn't documented anywhere that we know of; Hutchinson found it by accident. It works very well, at least on MBASIC 5.0.

The procedure is as follows. Suppose that you want to duplicate line 10, with the duplicate to appear as line 320. You give the command LIST 10. Immediately

after the line appears, enter a control-A. The program will respond with the exclamation mark prompt. Enter an "I" followed by the target line number ("I320" in this case) and press return. The new line 320 will be displayed; it is a duplicate of the last line displayed by LIST. Hutchinson notes that this is especially handy when you want to make an LPRINT that is a duplicate of some earlier PRINT statement. We played around a bit and think that there may be more abilities hiding behind this control-A input.

Etch-A-Vector

In the December *DDJ* there appeared an article by Marek W. Michalski, "A Simple Vector Generation Algorithm." Michalski provided a flowchart and gave the math background of the algorithm, but didn't display a program. We couldn't resist the challenge of making a program from his algorithm, just to see if we really understood it. The only graphics machine handy was an Atari 800, so we wrote the program in Atari BASIC.

It turns out that Michalski's algorithm is indeed a neat one. In coding it, the challenge comes in finding a way to fold the eight possible cases of vector direction into as few code routines as possible. The eight cases arise from the interaction of three factors. The X-direction may be positive or negative; the Y-direction may be positive or negative; and the vector may be longer in the X-dimension or longer in the Y-dimension. There are eight combinations of those three choices. The flowchart given in the article covered only one case, that of X positive, Y positive, and DX greater than (or equal to) DY. As it turned out, only two routines are needed to handle the eight cases.

The program (Listing One, this page) runs on an Atari, but doesn't have much use there because Atari BASIC already has a vector statement (DRAWTO). But the program could easily be converted to another BASIC; only the Atari verbs GRAPHICS, COLOR, and PLOT would have to be changed. More to the point, the algorithm does look as if it would be very fast in machine language (just as Michalski said). Can someone use it that way? Would someone like to put it into 8086 code for the IBM PC?

Our recent touching pleas for input have produced some good letters. Thanks to all who wrote, but please don't stop writing; this column devours material voraciously. We are in perpetual need of problems, pitfalls, puzzles and eurekas.

DDJ

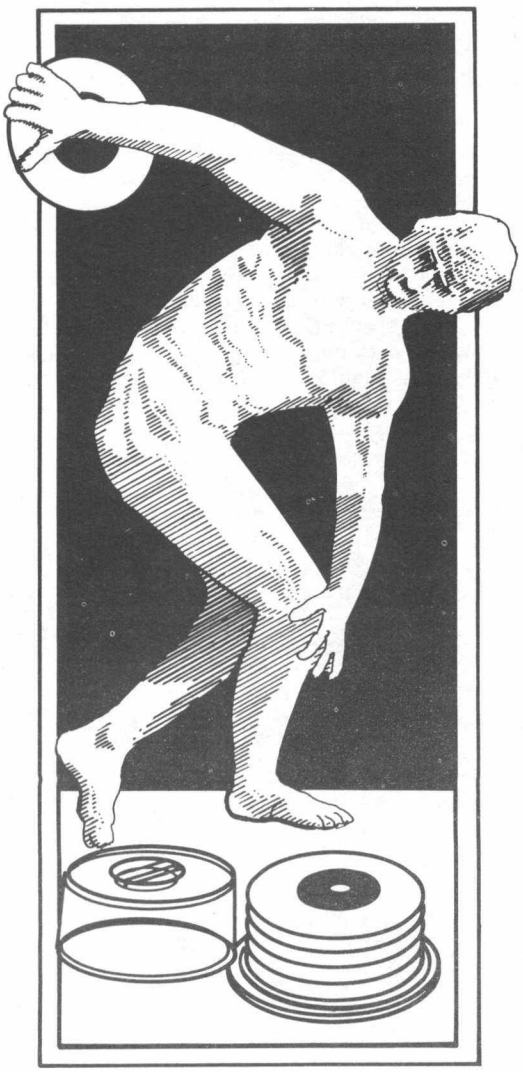


Illustration by Al McCahon

```
100 REM ************************
110 REM FAST-VECTOR ALGORITHM BY
120 REM MICHALSKY, DDJ #74 12/82
130 REM ************************
140 GRAPHICS 7:COLOR 1:REM ATARI ONLY
150 REM ***** acquire end-points
160 REM ATARI now set at 160h x 80v
170 ? "Start at (x,y)";:INPUT XA,YA
180 IF (XA<0) OR (YA<0) THEN 170
190 IF (XA>159) OR (YA>79) THEN 170
200 ? "Ending at (x,y)";:INPUT XZ,YZ
210 IF (XZ<0) OR (YZ<0) THEN 200
220 IF (XZ>159) OR (YZ>79) THEN 200
230 REM ***** initialize
240 X=XA:Y=YA:F=0
250 DX=XZ-XA:IX=SGN(DX):DX=ABS(DX)
260 DY=YZ-YA:IY=SGN(DY):DY=ABS(DY)
270 DX2=DX/2:DY2=DY/2
280 IF DX<DY THEN 410
290 REM ***** case of DX>=DY
300 REM step X by ones, Y as needed
310 PLOT X,Y
320 X=X+IX:F=F+DY
330 IF (F<0) OR (F<DY2) THEN 350
340 F=F-DX:Y=Y+IY
350 IF ABS(XZ-X)>1 THEN 310
360 PLOT X,Y:GOTO 170
390 REM ***** case of DX<DY
400 REM step Y by ones, X as needed
410 PLOT X,Y
420 Y=Y+IY:F=F+DX
430 IF (F<0) OR (F<DX2) THEN 450
440 F=F-DY:X=X+IX
450 IF ABS(YZ-Y)>1 THEN 410
460 PLOT X,Y:GOTO 170
```

Listing One

CP/M-80 Expansion Card For The Victor 9000

Product: CP/M Expansion Card for The Victor 9000 and IBM PC
Manufactured by: Small Systems Engineering, 1056 Elwell Court, Palo Alto, CA 94303
Price: $595 for 4 MHz version
$650 for 6 MHz version
Reviewed by Hank Harrison

Last year a British corporation designed a board dedicated originally to the Victor 9000 version released in Europe under the name "Sirius One." This little board is a bargain and is the first real add-on available for the slowly but surely developing Victor 9000 machine—a machine which now comes supplied with 10Mb hard disk for under $6,000.

The Board is a Z-80 based expansion board designed to make most CP/M-80 software available for the Victor-Sirius. It is also available for the IBM PC and is similar in both machines except that the 6MHz model costs about $55.00 more than the 4MHz version.

Bear in mind that Chuck Peddle's Sirius system was available in Europe one full year prior to the IBM and that most European designers use the Sirius as a standard, not the IBM. Frankly, the Sirius is, at least at this point, twice the machine for the money as it comes out of the box.

I am difficult to impress, but this board, one of the few expansions available for the Sirius-Victor at this time, is very impressive and is valuable for anyone developing software for the rugged and well-designed Victor 9000. If you are not developing software for it, and are among the hordes that have flocked to IBM, I strongly suggest you investigate this classy underdog.

Price and function are the two main variables I look for in a piece of hardware. Next, I consider reliability and ruggedness After these three physical parameters, the real proof of the pudding comes to play: I am interested in creativity and expression, not mnemonic frustration or difficulty disguised as "challenge." I will, therefore, consider any board, any chip, any piece of software that helps me get to my ultimate goals—absolute transportability and absolute transparency. It is obvious that Peddle *et al.*, intend to hold their market loyalists through quality and durability and although not totally transparent, the Small Systems Engineering Z-80 board is a definite step in the right direction.

After a while a reviewer, tech writer or product analyst gets a good feel for what's good and what's slipshod. Shoddy workmanship is very obvious in any computer product, overt cosmetics are also obvious. Sometimes I really wonder where the design engineers went to school. I wasn't aware Dior held classes in hardware these days but perhaps they should.

I have seen all manner of boards, tracings, jumperings and reconfigurations over the past seven years. IMSAI, for one, spent minimal time in R and D and documentation, almost everything went into production as soon as possible. The boards just went out into the field and the field debugged everything. In those days people were happy to do it for Apple, MITS or IMSAI because almost everybody was a "freak." This is no longer the case – even the most dedicated computer heavies don't have time to "mess about."

Esoteric products do not appeal to the New Technology user. People don't really want to modify their systems. Chuck Peddle knew way back in those Halcyon days at Commodore that the future big bucks were wrapped up in an "out-of-the-box" system. The Apple IIe is following suit.

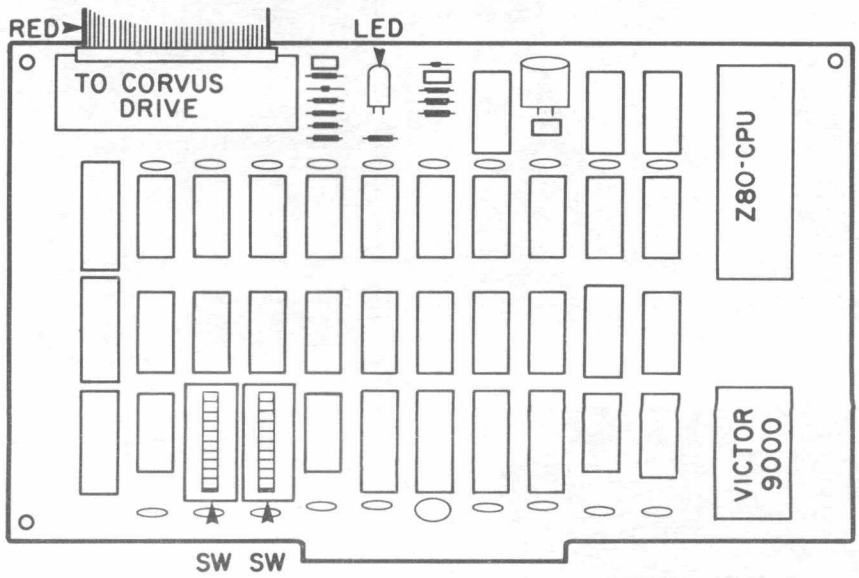
The peripherals business, which is where the big money is now, is thriving because the out-of-the-box system is often anemic, incomplete and oblivious to things like hooking up modems and printers. This is not the case with the Victor.

Unlike Americans who have been overwhelmed with marketing blab, Europeans, who see very few ads on television (and are therefore deprived of Charlie Chaplin's antics in the bakery), prefer quality and completeness out of the box. Europeans tend to be penny pinchers who demand portability of their old Software and it was to meet this need that the Small Systems Z-80 card was born.

The Small Systems Z-80 card is a multi-function wondercard. It plugs in and that's it. Installation in my system took twenty minutes, five of those minutes were spent cleaning the screen since I had the monitor off anyway.

The card fits any of the four Victor 9000 expansion slots. We selected number one as it is closest to the venting holes.

The card must be fitted with the chips away from the disk drives, but there are no red arrows printed on the card. There is also no way to cement this in place and the bus structure is almost like an S-100 system with no top clips or slot guides. I strongly urge Victor Systems to build a $12.00 add-on cage frame or Small Systems to provide clips.



Small Systems Engineering expansion card for Victor 9000 adds 64K RAM, hard disk, virtual disk, and BSTAM capabilities.

I feel very weak-willed in moving a computer from room to room when the boards are rattling in the slot. I have invested more than $1,500.00 in custom "suitcases" for my system. The Victor is very portable in this fashion and weighs about as much as two Osbornes or one KayPro II and a printer, but no amount of outside protection can compensate for a kludgy bus connection.

Once in place, the board achieves some small miracles:

1) It gives 64K of extra RAM and boots as a separate computer.
2) It boots CP/M-80 programs.
3) It connects to Corvus or Corvus-like 5, 10 and 20 Mb hard disk units.
4) It makes possible BSTAM, or "move it," communications.
5) It utilizes Corvus networking or any networking conforming to Corvus protocols.
6) It supports diagnostics, e.g., Memtest, DDT, etc.
7) It allows for Corvus Mirror back-up, although floppy front-up, if done properly, is all that is needed.
8) It provides an "M" option which is a virtual disk that can be written to and which can act as an executive.

For the most part the ability to run more than 2000 old programs still good enough to use on a hot new 16-bit system, makes the board worth the price, but this board also takes advantage of the large amount of RAM available in my system, presently 256Kb plus the 64K on the 8-bit Z-80 card.

Imagine the contradiction in terms: a CP/M-80 file in excess of 20K safely loaded into RAM and processed by WordStar with no disk access after boot. In fact, my book chapters are typically 90K. Using the Small Systems card, I was able to use a footnote, a Thesarus, and a global search and replace, in both directions with no penalty and with amazing speed. I also booted a German data base called Aladdin and my own proprietary spelling checker called Zero Glich. I experienced no "DELETING CURRENT BACK UP TO MAKE ROOM FOR SAVE" error messages.

We will be continuing to review the Victor developments. Unfortunately, we just didn't have time to compare the Small Systems Z-80 board in both intended hosts at this time. I personally would love to do a straight head-to-toe face-off comparison between IBM and Victor, but many factors mitigate against this.

In short, the Small Systems Z-80 board opens up an entire range of micro capabilities from 8-bit to potential 16-bit systems and makes the Victor 9000 quite attractive, especially in view of Chuck Peddle's recent *fait accompli* wherein the entire operation is coming back home from Chicago. Peddle wants to locate the French headquarters in Nice.

The following information represents an update on the Sirius-80 card, and on applications not currently covered in the User's Manual.

Hard Disk

There are now two ways to operate a hard disk on the Sirius-Victor using the Z-80 card. First, under CP/M-80 which runs on the card's Z80 processor and can access the card's built-in Corvus hard disk port or alternatively under CP/M-86 which runs on the Sirius main processor (8088) and uses the Sirius-80 card as a hard disk interface.

The hard disk interface on the Sirius-80 card will drive a Corvus 5, 10 or 20 megabyte hard disk but others may soon be plug compatible. CP/M operation using a hard disk is similar to operation using floppy disks, but is much faster and offers more on-line storage capacity.

Drive Access Time:

10 or 20 Mbyte drive:	80ms maximum 40ms average
5 Mbyte drive:	240ms maximum 125ms average

Normally drive A will be the handle hung on the disk drive, because it will be used most. As CP/M version 2.2 normally only supports drives up to 8 Megabytes. The drive is divided into two or four logical CP/M drives. These will typically then be:

5 or 10 Megabyte drive –
CP/M drives A: and B:

20 Megabyte drive –
CP/M drives A:, B:, C: and D:

"Drive M" is an additional feature which is built into the latest release of the Victor-80 card software. It allows use of any additional RAM cards in the system by treating them as a CP/M pseudo disk drive, called drive M. This enables the loading of all frequently used files and programs into RAM at the start of a session (by PIPing them across onto drive M) for faster execution.

The RAM card behaves as a high-speed, volatile (but otherwise normal) CP/M disk drive, and all the normal commands such as STAT, PIP, ERA and DIR will work on it.

"Drive M" Operation Under CP/M-80

The System Disk is supplied with a file called NEWSYS. Typing in NEWSYS and a carriage return brings it up. The NEWSYS program will ask:

Reconfigure "80.CMD" on which drive (A-P)?

Reply to this question with:

A

followed by a carriage return.

When the system to be reconfigured has been loaded into memory, NEWSYS will display a menu of options:

A – autoload command
D – disk drive assignments
P – change I/O port number
E – execute reconfigured system
S – save reconfigured system
Q – quit to CP/M

Typing "D" followed by a carriage return, you now have seven choices of disk drive assignments as shown in Figure 1.

The drive M parameters for CP/M-80 can be changed by using the command in the NEWSYS program. The first parameter is the base starting paragraph of the memory to be used by drive M (i.e., the hex starting address divided by 16). This defaults to 2C0 hex, the first memory paragraph not normally used for screen handling. The default may be altered if special screen handling routines are used or if drive M is to be used with a RAM card which is not contiguous with the normal system memory (e.g., to maintain compatibility with drive M as used under CP/M-86).

The second parameter is the memory size (number of Kilobytes, in decimal) to be used by drive M. This would normally be set to the size of the memory card, or zero (to automatically use all the memory available up to the base of the operating system software).

The final parameter specifies whether the drive M directory is to be cleared on entry to CP/M-80. This would normally default to YES unless it is desired to access the same memory card from both CP/M-80 and CP/M-86. In this case, the directory will be cleared once and for all when CP/M-86 is booted, and all files will be preserved and can be accessed under both CP/M-80 and CP/M-86 until the system is powered down or reset.

If the question, "Do you wish to use the 'Drive M' RAM card facility (Y/N)?" is answered with a "Y", HARD86 will ask for the start address and size of the RAM card. The start address should be a hexadecimal paragraph address (1 paragraph = 16 bytes), and the size should be the decimal number of Kilobytes (multiples of 1024 bytes) so that if the RAM cards (or card) occupy physical addresses 80000 to BFFFF hex, enter "8000" for the start address and "256" for the size.

Memory blocks can be addressed anywhere in the range 20000 to DFFFF hex but memory cards should be discontinuous with the main system memory

(at 00000 to 1FFFF hex). This tidiness prevents CP/M-86 from relocating itself into free memory.

Remember, any files stored on drive M will be lost on cold boot and files on drive M will be preserved during operation under CP/M-80.

Note also that CP/M was never originally designed to operate in a multi-user environment. MP/M was designed to support more than one user *on the same processor*, but this involves sharing of the central processor and memory as well as a disk drive. The Victor-80 card remains a single user system enhancement and is not a co-processing card, but allows networking through a constellation box.

Input/Output Addressing

The Victor-80 card occupies one of the 65,536 available Victor I/O port addresses. It is normally shipped to work with I/O port 0, but this may need to be changed if it conflicts with other expansion cards in view of the fact that a full Megabyte of RAM is now available.

To change the port address, you will need to alter the switches on the Victor-80 card and change the port address in the CP/M-80 or CP/M-86 system software (using a NEWSYS or HARD86 program respectively).

The following switch assignments default to OFF and refer to the two on-board Dual In-line Packages. The listing implies ON is selected.

Switch 1 (nearest center of card)

1 – I/O address bit 0
2 – I/O address bit 1
3 – I/O address bit 2
4 – I/O address bit 3
5 – I/O address bit 4
6 – I/O address bit 5
7 – I/O address bit 6
8 – I/O address bit 7
9 – I/O address bit 8
10 – I/O address bit 9

Switch 2 (furthest from center of card)

1 – I/O address bit 10
2 – I/O address bit 11
3 – I/O address bit 12
4 – I/O address bit 13
5 – I/O address bit 14
6 – I/O address bit 15
7 – Reserved for future use
8 – Reserved for future use
9 – Reserved for future use
10 – Reserved for future use

7 and 8 should be left unaltered.

The Z-80 card address map appears as shown in Figure 2.

Some Notes on Serial Communications – BSTAM using Serial Port "B"

A version of the BSTAM driver has been produced to use serial port B on the Victor 9000. This allows serial port A to be reserved exclusively for use with a serial printer (the TTY: device), the parallel port is usually designated for a daisy wheel, the serial for dot matrix, but a switching set-up on the parallel port is highly recommended for multi-printer installations.

The new driver is in a file called USIRIUSB.ASM on the Victor-80 disk. It should be assembled and patched into your copy of BSTAM described in the User's Manual, Appendix A.

Note that both the "port A" and "port B" BSTAM drivers automatically pre-set the appropriate port to 9600 baud. If the machine being communicated with will not work at 9600 baud, you will need to modify the Victor-80 BSTAM drivers to set up a different baud rate.

The Victor 9000 serial port is wired to appear as a terminal for use with modems and for local, double twist wire networking. Thus pin 2, instead of the more usual pin 3, is used for handshaking. This means that when using BSTAM to connect the Victor to another machine, pins 2 and 3 should *not* be crossed over in the connecting cable as they normally are in the RS232 tradition. The cable connections should be:

Computer		Remote computer
pin 2 (data out)	→	pin 2 (data in)
pin 3 (data in)	←	pin 3 (data out)
pin 7 (signal ground)	↔	pin 7 (signal ground)

It is important to ensure that the copy of BSTAM on the remote computer has been correctly configured for that machine, and that the remote machine has been set up for the correct baud rate (i.e., 9600), *one stop bit, eight data bits, and no parity.*

Setting Up Port A for a 9600 Baud Serial Printer

The program "9600.CMD" on the Small Systems Z-80 diskette is provided to set the CP/M list device to be TTY: (serial port "A"), and at the same time, set its baud rate to 9600. This program is necessary because the baud rate setting program provided by Victor has a current maximum of 4800 baud.

In the final analysis, things are brewing at Victor, that small think-tank up in Scotts Valley just beyond the Silicon smog margin. And sales at Small Systems Engineering should heat up rather quickly. It is hoped that multi-purpose boards like this will be available for all computers so we can all move anywhere we want.

DDJ

SOFTWARE REVIEW

REVAS Disassembler

Product: REVAS Version 3.0
Company: REVASCO, 6032 Chariton Avenue, Los Angeles, California (213) 649-3575
Computer: Z80 CPU with CP/M
Format: 8" SSSD, SSSD Northstar, Microplis Mod II
Minimum Memory: 32K
Release Date: January 1982
Price: $90.00
Reviewed by Steve Willoughby

This disassembler produces console, printer, and .ASM and .PRN type files from 8080 or Z80 machine code which is memory resident or disk resident in .COM type files up to 64K bytes in length.

The disassembler will disassemble the undocumented Z80 opcodes in addition to standard opcodes. Output mnemonics are in TDL or ZILOG format and are selected by running the particular variation of REVAS (both included).

The disassembler provides locking of specified memory areas for disassembly as BYTE or word data. Provision is also made for multiple inclusion of a comment line in the output as a particular location is disassembled.

The disassembler includes search and cross reference functions, display of internal and user parameters, and a calculator mode. Numbers can be input and output in any radix with independent input and output radices. Provision is made for user parameters to allow customization of print output with many popular printers.

BACKGROUND

A disassembler reverses the process in which an assembler converts mnemonic codes and symbolic values into machine language. A disassembler, therefore, attempts to reconstitute the mnemonic code from the machine code. The disassembler is used to maintain or modify machine code for which the sources are not available, and as a debugging tool. Those who have used DDT have experienced a limited disassembler.

Several problems prevent the disassembler from performing the ideal translation of the machine code into the original source code. The original assembly removed the source comments and converted the symbolic arguments into absolute binary numbers; in addition, the disassembler cannot distinguish embedded data and instructions. A good disassembler allows the human to interact and remedy these problems. The human can disassemble an area of code looking for meaningless instruction sequences or blocks of data which have ASCII values reading as text. These areas can then be flagged as embedded data and will be subsequently decoded as data, not instructions. A good disassembler will generate synthetic labels wherever a number would have been generated from a symbolic argument.

An outstanding disassembler builds a symbol table of these synthetic labels and allows the human to redefine the label to one which is meaningful (or the original symbolic label). Further disassemblies will replace the number with the label from the symbol table. This provides disassembly listings that look very much like the original source code. An outstanding disassembler allows comments to be added to the output. While size and memory usage constrain the degree of comments available, at least there should be provision for inserting a line of comments at key places in the disassembly to document program sections or subroutines.

A good disassembler will disassemble memory resident code and produce a printed output. An outstanding disassembler accepts disk file input and output in addition, allowing a very large program to be disassembled and producing output on a medium ready for re-assembly.

A good disassembler allows aids for the human to calculate binary add and subtract in one radix (usually octal or hexadecimal). An outstanding disassembler provides for dynamic specification of radix and also provides multiplication, division, and modulo calculation.

The general usage of a disassembler follows this scenario: User has machine code program for which vendor won't release source code and which has some bug or feature preventing operation on user's system; or software won't work with user's particular peripherals; or the user has printed source listing but no computer readable form of sources. User first uses disassembler to generate preliminary source file. User then uses editor to make changes, add comments, etc. User finally assembles source file to generate new machine code program curing problem or proceeds with edit-assembly-try iteration until problem is solved.

REVIEW

As one may gather from the previous discussion of symbolic labels, data and comments, one does not just dash off a disassembly, even with an excellent disassembler. The capability of the disassembler, however, will affect the time required to get the desired results and the sophistication of the results (you get what you pay for).

I've used the REVAS Version 3.0 disassembler and consider it outstanding in all respects. REVAS is very complex because of its many features to help the human decipher the machine code; however, the manual has a guided tour through a disassembly of a supplied disk file that quickly and effortlessly acquaints the new user with REVAS to the point of self-sufficiency. I appreciated the calculator mode's ease of converting disk block numbers to sector and track using the divide and modulo fuctions. I used REVAS on a CP/M 1.4 system so REVAS should also work for those who are running CDOS.

The manual is near perfect. In fact, I found only two errors in the manual. The manual was well-organized and well-polished, evidencing its preparation on a word processor with a daisy-wheel printer.

My two complaints are that the copying is uneven, showing it was not typeset, and that the general tone of the introductory chapters seems to talk down to the reader making reading tedious. My overall impression is that of a very mature product.

My resistance threshold on software of this type is $50, but I will make an exception for REVAS. It is definitely worth its $90 pricetag to someone who needs more than the disassembly capability of DDT or DEBUG. My next project is to convert a cassette BASIC to run under CP/M. This is the sort of project for which REVAS is ideal.

Performance: Excellent
Documentation: Superior
Price: Good
Value: Excellent

DDJ

BOOK REVIEWS

Data Communication for Microcomputers
By **Nichols, Nichols, and Musson**
Published by **McGraw-Hill**
$16.95
also
TRS-80 Data Communications Systems
By **Frank Derfler, Jr.**
Published by **Prentice-Hall**
$12.95
Reviewed by **Joseph B. Rothstein**

The computer has been called "the general purpose tool," and one of the most important questions any computer designer or computer user must answer is "What does a general purpose tool look like?".

One answer to that question, which was not anticipated by the first computer engineers, is that the modern computer often looks like a telephone. Numerous computer-based products exist solely for the purpose of communicating over telephone lines, while an equal number of products serve as add-ons to conventional computers, enabling them to tap into the growing number of databases, information utilities, and other users accessible through the worldwide telephone network.

But the microcomputer user anxious to enter this brave new world is confronted by a bewildering array of buzzwords and options. Having only begun to master the jargon associated with microcomputers, the neophyte may be reluctant to face yet another learning curve, characterized by such terms as "physical layer protocol," "star topology," and "collision handling."

Two recent books with similar titles attempt to address this dilemma, each in a different fashion. *Data Communication for Microcomputers*, by Elizabeth A. Nichols, Joseph C. Nichols, and Keith B. Musson, offers a wide-ranging discussion of data communication principles and hardware, without focusing on a particular computer. The authors are consultants in the data communications field, and their text might be more appropriate for the curious user of a large-scale telecommunications system than for the first-time microcomputer owner who only wants to know how to dial up one of the consumer-oriented information utilities.

TRS-80 Data Communications Systems, by Frank Derfler, Jr., focuses instead on a much narrower topic – the implementation and use of telecommunications systems for Radio Shack's popular microcomputer.

Both texts address the essential question, "What is data communications, and how can I make use of it on my microcomputer?". But beyond that, they are hardly equivalent treatments of the same subject.

The Nicholses and Musson approach the subject from three perspectives. Their intention is to offer practical information on such topics as how to fabricate a cable or choose a modem, while at the same time providing a tutorial on the concepts and principles of data communications and also serving as a reference tool. They are partially successful in meeting each of these goals, but despite the text's 264 pages and hefty price tag, I was not fully satisfied with their handling of any one of their three areas of concern.

The hardware hacker, who appreciates workshop tinkering for its own sake, may find the most to enjoy. The authors have included a series of experiments for the reader to perform, ranging in complexity from fabricating a three-wire economy model RS-232-C cable, to implementing interrupt-driven I/O on an S-100 or TRS-80 computer. Though the experiments are thoroughly described, including lists of required materials and equipment, as well as software listings, the relationship of the experiments to real-world data communications tasks is not clearly established. Several of the experiments involved fabricating different kinds of cables, for example, yet the related text does not adequately make clear how the user is to know what sort of cable is required for a particular situation.

Novices to data communications may find themselves overwhelmed by the tutorial on the subject, for it includes far more than most microcomputer owners would find necessary in order to join the ranks of computer communicators. While there's surely nothing wrong with offering more information than required, many readers may become impatient with an approach that does not get around to covering modems until page 221.

Though there's no lack of information in *Data Communication for Microcomputers*, it falls short of being an important reference on the subject. There are no appendices, quick reference guides, or comprehensive tables, so random access to a particular topic is limited to the table of contents and the index. Telecommunications managers would probably not want to rely on *Data Communication for Microcomputers*, but it might be useful to a business person trying to cope with the jargon used by data communications salespeople.

Perhaps the authors would have been more successful in their intentions if they had actually written three volumes, instead of trying to satisfy three audiences in a single text. By trying to be all things to all types of readers, they have limited their usefulness to each.

This is not a problem for Frank Derfler, Jr. He has the audience of *TRS-80 Data Communications Systems* firmly in mind on each of the 159 pages, and goes right to the heart of the subject. The role and fundamental principles of data communications are covered in the first twenty-one pages, while the bulk of the remaining text is devoted to the implementation and use of the TRS-80 as a data communications instrument. Beginning with a discussion of the TRS-80 serial port, Derfler covers such topics as modems, terminals, accessing microcomputer-based message systems, and telecommunications for the deaf.

Of course, much of the text is of limited value to owners of other brands of microcomputers, but that is to be expected. The emphasis here is on the practical needs of the TRS-80 user, and this audience is well served. The author's approach is measured and thorough, though not at all dry or technical. Practical considerations are emphasized in discussing such subjects as the difference between "originate" and "answer" mode, or how to determine which signal goes on which wire of a cable.

The text includes descriptions of specific products available for the TRS-80, and an appendix contains the names and addresses of more than 20 manufacturers and distributors of data communication products. Similarly, the author mentions several information utilities and microcomputer-based message systems of potential interest to TRS-80 users. And unlike the Nichols/Musson text, there is a brief glossary of important terms.

Frank Derfler, Jr. writes frequently on data communications topics for the microcomputer popular press, and he uses the breezy, conversational style of those publications in his book as well. His observations on such topics as the current information network, scenarios of the future, and social analysis contribute to the readability of *TRS-80 Data Communications Systems*, and provide food for thought which serves to complement the

"how-to" aspects of the text.

The TRS-80 user who wants to cross the frontier of data communications would likely have a long wait for a more valuable text than *TRS-80 Data Communications Sytems*. Others, using different gear, may find *Data Communication for Microcomputers* worth considering as an interim step, until something more appropriate and useful comes along.

The Network Revolution: Confessions of a Computer Scientist
By **Jacque Vallee**
Published by **AND/OR Press, Inc., Berkeley, California**
$7.95
Reviewed by **Virginia Lyons**

For those readers who enjoy sociological and philosophical speculations on computer systems, Jacques Vallee has just published his new book, *The Network Revolution: Confessions of a Computer Scientist*. In it Vallee outlines just how communications are changing the fundamental structure of modern society.

Until the development of the personal computer, mass communications were centralized, one-way, non-adaptive messages distributed to a captive audience. Television, telephone, radio and print media are advertising and marketing industries in which the mass audience is the product sold to the advertiser. Messages of the mass media are based on the industrial organizing principle of mass distribution. This order has endured because there has been no demand for an alternative. Because people did not have access to the media, they could not disseminate alternatives to a mass audience. Dr. Vallee argues that mass media stems from it's control by sectarian interest groups which avoid two-way communications.

Now, two-way mass communication is possible within the context of today's technology. Such a communications system exists because inexpensive microcomputers and ultra-fast communications channels operating in real time are now available.

Vallee poses a choice between the "Digital Society" versus the "Grapevine Society." In the Digital Society, large computer systems are repressive devices which reduce people to statistics. Usage is highly restricted to governments and corporations without any public access. The Grapevine, however, allows people to build networks configured around personal computers, new cable TV services with access to data bases and information services. More importantly, these systems will be based upon the users' own interests. Vallee projects that these networks will become gateways to other minds, windows to other landscapes.

Vallee began his career as a computer scientist tracking Sputnik's orbits on a used IBM 659 in Paris. He subsequently participated in the design and development of computer networks. This background provides many clever anecdotes for a shrewd appraisal of human interaction with computers.

Vallee traces computer development from the work of Zuse and other German scientists prior to World War II through today's familiar and ever more present personal computers. When the billion-dollar ENIAC suddenly failed, the problem was traced to a dead moth trapped in a relay. This event gave rise to the word "bug," meaning "error." He also recounts the humor at meeting the person who first suggested that perhaps computer programs needed a "stop" command, something we take for granted now.

Of course, Vallee does not spare us from some of the more conspicuous time wastes. During the fifties, university computing departments volunteered to assist the Pentagon in the translation of voluminous amounts of Russian scientific literature. But, alas, all those millions of dollars were actually spent on retranslating tons of Rand Corporation and similar "think tank" publications which the Russians had translated from English!

Vallee gleefully goes into great detail about the failure of Management Information Systems, MIS. He contends that they were designed for the purpose of making rational decisions about the economics of running a business. However, in the real world, business is not run on a rational basis with the luxury of scientific tests. Managers make decisions under great stress and uncertainty. Their decisions are often based upon esthetic and intuitive responses. His experience of upper management was of elegant executives discussing Sartre and Nabokov over tea.

Surely the most interesting thing about computers is the variety of human responses. Vallee takes a closer look at the nature of information and how we humans relate to it. Computers do not actually store "information," but rather "data." Information is a subjective quality dependent upon who is examining the data. Data becomes information only after someone asks a question about it. Vallee draws the analogy of the doctor, nurse and hospital administrator reviewing a patient's records. Each person recognizes only specific parts of that "data" as "information." In this context, it is easy to recognize a database as a power center. Vallee contends that an information center is primarily a social system. It is the responsibility of system builders and users to humanize these machines. Too often, computer scientists have built systems which required "well-trained humans."

Vallee tells an anecdote of a visit to Russia. He carried along a stack of IBM cards coded into a simple program which might generate a dialogue with the Russians. Much to his amazement, he discovered that the Russians were more interested in the cards than the program. Although the Russians used the same shape and size of cards, each of the 80 columns was punched to maximum efficiency. Each column had more than one character. So the Russians could never tell visually what information or errors appeared on the card. Vallee makes the point that extending technology to "rational" limits completely leaves out the human element.

Somewhere in his career, Vallee realized that artificial intelligence had a fundamental fallacy. Artificial intelligence tries "to emulate nature, it wants to approximate what Man does. Imitation of nature is bad engineering." Man does not fly by emulating nature and strapping on wings. Man flies by putting a 400 horsepower engine on a door. Nature never created the Boeing 747, because a creature that flew 700 mph at 40,000 feet was not needed. "How would such an animal feed itself?"

Vallee questions the very nature of intelligence. Perhaps Mother Nature "has never needed an intelligent animal and accordingly, has never bothered to develop one." When a truly intelligent machine is built, it will be based on principles different from Man's mind.

Vallee delights in satirizing various systems developers. When a user confronts a programmer with a bug, a likely response is a "fix-up" with another command. Later, the programmer attends computer conferences and extols the "power" of his system. Vallee prefers to take a large system and "sculpt" it down to a facile and "friendly" size. He also pokes fun at all the silly things designers have done with computer keyboards.

"Grapevine" networks based on phone lines and microcomputers are a subject dear to Vallee's heart. He gives us a mini-history of computer crime and phone phreaking. He visits Chip Tango, a teenage system "cracker." Chip introduces Vallee to the "underground network." No longer do people conference over private networks about "bytes and bits." Areas of interest now range widely over topics such as gay sex, music, NASA, jokes, gardening and software piracy. As wonderful as such networks appear, Vallee points out a few hidden dangers. If every home had a terminal, the phone lines would be rapidly overloaded. Ma Bell never designed the phone system for such an interaction, and cable TV reaches only about 20 percent of American homes. Moreover, it is unlikely that Americans will ever be able to use cable in a truly two-way interactive system. Vallee also fears that such a worldwide communication system may accelerate an

already deteriorating sense of trust in society.

Lest readers of this review get the impression that Vallee's sense of the "Grapevine Network" is all gloom and doom, relax. In the final chapter, he meets Dr. Breeze, a Washington bureaucrat who turns into a gnome. Lovers of adventure games will delight in this encounter. Dr. Breeze makes some interesting observations about ants, and he grants some mighty powerful wishes.

Readers who would like a first-hand accounting of the history and development of computer networks will enjoy this highly amusing book. Jacques Vallee certainly focuses on the relevant sociological issues of network systems.

The Online Micro-Software Guide and Directory 1983/84
Edited by **Helen Gordon**
Published by **Publications & Conferences for the Information Industry, 11 Tannery Lane, Weston, CT 06883**
$40.00
Reviewed by **Hank Harrison**

We always look forward to the arrival of the *Online Directory*. This reference work, although expensive, is of great value to anyone working in the reference end of the software business. It is not only a sort of Whole Earth catalog of available software for all available systems, but it is an education in itself. It is in fact a data base for software and hardware.

The *Online Directory* was edited specifically to meet the needs of librarians and information managers. It has feature articles on choosing terminals and microcomputers. Terminals are handled by Janet Bruman, who handles terminal evaluation for the California Library Authority for Systems and Services (class) and Dennis Cagan, President of the David Jamison Carlisle Corp., one of the largest American equipment brokers. The microcomputer selection guide is by John C. Blair Jr., Computer Applications Librarian at Texas A&M Medical Center and author of the micro columns in both the *Online* and *Database* magazines.

The directory also contains equipment specifications for over 60 teleprinter terminals, 70 video terminals, 30 printers, 40 modems and over 65 microcomputers, plus a data base comparison chart.

The sales and service office portion of the directory is an alphabetical listing of over 1500 service agents both U.S. and international.

The major criticism of the catalog is its thrown-together appearance. For that price a bit of Lettraset could go a long way. The major titles and catagories are not boldfaced and it does require some plowing through.

CRUNCHERS-21 — Simple Games for the Timex/Sinclair 1000 2x
By **Yin Chiu and Henry Mullish**
Published by **McGraw-Hill**
$8.95, 137 pp.
Reviewed by **Trina Watson**

This little gem is an ideal small education book, although priced a bit too high in this reviewers opinion.

CRUNCHERS contains twenty-one original games programmed in the T/S 1000's version of the Basic language. Not only are the games intriguing and entertaining, they also provide an informative introduction to general programming, which is the main attraction of the book, and all of the programs work.

All twenty-one games are fully explained for the absolute beginner. For the advanced reader there are suggestions of how to modify and enhance the programs. Explicit instructions are given on entering, editing and running programs, saving programs on a tape recorder and loading programs into the computer from tape. The plastic comb binding ensures that the book will be open flat on a desk, a major convenience for a person who is entering the programs into the computer.

The experience gained in becoming familiar with these programs should be able to be put to direct advantage in regular programming situtations.

DDJ

OF INTEREST

by Michael Wiesenberg

Trends

Did you see Hewlett-Packard's announcement of its new 32-bit superminis, the **HP 9000** family? Mainframe power at each networked workstation (Ethernet-compatible and able to communicate with other families of HP computers, like 3000s, 1000s, 125 series, and the new 200 series), using UNIX or BASIC operating systems. With built-in 10 Mb Winchester and printer, 256K floppy drive, color or monochrome graphics CRT, the system fits easily on a desk-top, or is available in rack-mountable box or cabinet. Offering up to three CPUs, languages including C, FORTRAN 77, Pascal, macro assembler, and compiled BASIC, a symbolic debugger, engineering and scientific software, the 9000 costs from $28,250 to $64,565. A lot, you say? Yes, but the low-end VAX *starts* at $50,000 and the high-end starts at over $210,000. This information should interest *Dr. Dobb's* readers because it suggests that 32-bit systems will soon be available to personal computer users at reasonable prices. **Reader Service No. 101.**

A New York City bank has offered ATARI 800s as an inducement to savings instead of Crocker Spaniels and other fuzzy bank critters. It remains to be seen if **Dry Dock Savings Bank's** plan made them the largest retail distributor of home computers in the country over the Christmas season (the offer was good till the end of '82), as they claimed. Either deposit X thousands of dollars and get an Atari Education and Entertainment System or Home/Office Management System in lieu of interest, or buy a system at reduced price. Atari has more information on this. **Reader Service No. 103.**

Fight Self Pollution

Don't let your system pollute itself! (Electrically, that is.) Eliminate AC power line "pollution" (noise, hash, and surges and spikes from lightning and nearby heavy machinery) with the **Magnum Isolator** from Electronic Specialists, Inc. Each of the four "individually quad-Pi filtered" AC sockets handles a 1000-watt load, or 1875 watts on the entire device. $181.95. **Reader Service No. 105.**

Power to Spare

For ultimate safety, you'll want **Standby Power System** (SPS) from SAFT America. The Model SPS0200 battery/inverter system supplies 200 VA of emergency AC power at a nominal 117 volts for 20 minutes, and the Model SPS0400 supplies 400 VA for 10 minutes. It cuts in within 4 to 6 milliseconds (within a half cycle) whenever line voltage drops below 102. The SPS has an outlet signal to trigger software to start an automatic emergency shutdown, or safely store your data to disk and manually shut down your system until power is restored. Output voltage ranges from 100 to 250, and power output is 50 or 60 Hz. The SPS also protects against spikes and transients in voltage. Each unit is about the size and weight of a car battery. $600 for the SPS0200, and $800 for the SPS0400. **Reader Service No. 107.**

160 Mb Winchesters

Forget those disk drives that offer only five megabytes and cost $2495 and up. The **Disctron D-1600** from Disctron, Inc., is an 8-inch Winchester fixed-disk drive with *160* megabytes (unformatted), currently available only on the OEM market for $1980. Thin-film plating technology stores data on seven surfaces, each with a dedicated read/write head, of four fixed disks, with the eighth for track and servo information. Disctron also offers a 10.6 Mb 8-inch Winchester with removable cartridge and an 8-inch fixed disc with 42.4 Mb, both formatted, and 5¼-inch drives with 6 through 26 Mb (unformatted). **Reader Service No. 109.**

Sex-Change Operations

Reverse the gender of any RS232 line for compatibility between any accessories with **Gender Reversers** from B & B Electronics. Two models, one with two male and the other with two female ends, each interconnects all 25 pins, each for $19.95 postpaid (specify which), or both for $34.95. And, while you're at it, connect any two RS232 devices in any pattern with B & B's **Wiring Adapter**, that includes ten plug-in jumper wires, for $24.95. B & B also offers a free illustrated catalog of interface and monitoring equipment. **Reader Service No. 111.**

Undocumented But Well-Substantiated Rumor

You know that Atari 400 or 800 you got for the kids for games, and that only comes out at holiday time? Does it have a disk drive? Bring the "toy" out of the mothballs! I've just heard a well-substantiated rumor that you will soon be able to add a black box and a card for under $500 that will let you run CP/M and, of course, all the applications that run under CP/M.

Attachment to Your Rumor

And to get a disk drive for your Atari for $488, Percom Data offers the **AT-88**, introduced with the wonderful quote, "We've more than a fond attachment for your Atari." The single-density drive has 88 formatted K, its own power supply, and plugs directly into the Atari 400/800. You get the OSA/Plus operating system thrown in; the drive can also use Atari operating system software without modification. Percom has also lowered the price on its double-density drives from $799 to $699. **Reader Service No. 113.**

New S-100 System

MASTERMAX, from John D. Owens Associates, is a four-slot S-100 Z-80 system with two 8-inch floppies, a single-card computer with CP/M or TurboDOS. The four-channel DMA controller runs a speedy OS, while disk emulation allows quick file access. Up to four users can access the *same bus* and data bus, using TurboDOS multi-user OS, which is compatible with CP/M 2.2. You get a 4 MHz Z-80, 64K RAM with bank select, four I/O ports (two RS232 serial with one channel programmable in DMA, interrupt, or programmed mode, and two parallel), memory management of up to 16 Mb system memory, and eight vectored priority interrupts that chain with I/O interrupts to use the Z-80's interrupts. The standard drives are single-sided, double-density, but you can also get

double-sided, double-density; the disk controller handles both formats and controls up to four drives, 5¼- and 8-inch floppies or hard disk. **Reader Service No. 115.**

Software

Cloned versions of the two most popular video arcade games are available for CP/M from California Digital Engineering. **Catch-Um** features a dot-eater chased by vicious letters through a maze and **Ladder** sends you up ladders after a prize while dodging O's rolling down from above. Customize your keyboard to use any keys for up, down, left, right, and jump. You also get a personalization program to tailor the games in less than a minute to any terminal with direct cursor addressing. Each costs $24.95, or both for $39.95. Specify format. CDE also offers a wonderful combination word processor, text formatter, mailing label and database program for CP/M or North-Star DOS for $89, a library of Z-80 assembly language subroutines for $34.95, a debugging monitor for North Star DOS that examines and alters memory and registers, sets up to three break points, inputs and outputs directly to I/O ports, and renames files for $24.95, and Original Adventure for most systems (and the fastest-running available, they say) for $29.95. **Reader Service No. 117.**

Everyone's talking UNIX these days. We've seen UNIX-*like* operating systems for the IBM PC, but VenturCom's **Project Viking** offers one of the first "genuine" UNIXs for the PC. The Viking package includes Venix (a real-time superset of System III UNIX), the FinalWord word processor, Logix relational DBMS, and VenturCom's Graphix package. Developed on DEC's 11/23, and also available for the DEC VAX, PDP-11, DEC 11/2, Sony PC, and other 8086 computers, Viking costs $500 for the IBM PC. **Reader Service No. 119.**

Flying Ice Cream?

Vanilla Pilot is a full-featured Pilot language, including Turtle Graphics, for Commodore 4000, 8000, 9000, and Commodore-64. You also get a *Vanilla Pilot* editor to use with the computer's screen editor, adding commands such as FIND/CHANGE, TRACE, disk and cassette I/O commands, LOAD with append option, and RUN with load and execute option. The manual was written by "experienced educators and was carefully designed for clarity and easy reading." (So they say, but, as a technical writer, *I* say I'll believe it when I see it. And what are "experienced educators"?) Get Vanilla Pilot from your Commodore dealer or Tamarack Software for only $29.95. (Well, the price is right.) **Reader Service No. 121.**

Connections

Transfer files from your TRS 80 Model I or III to your IBM PC (you *do* have both, don't you?) with the communications package from **Personal Computer Products**. You get programs for both systems, including a verification program, and an adapter to connect the systems. Files can be of any length and can be concatenated. Transmission baud rates range from 110 to 9600. A user's manual contains, in addition to operating instructions, program conversion tips. All this for $39.95. **Reader Service No. 123.**

The **Apple-IBM Connection** from Alpha Software Corporation transfers information between (what else?) Apple II and IBM PC. You get a floppy for each machine, and connect by direct cable or modem. Transfer rate is up to 100 characters per second. Either machine operates in "master" or "slave" mode, so you can control communications at either end. With an automatic answering modem, you can make a long-distance transfer over the phone lines by yourself. Error-checking procedures ensure integrity of the data. You can transfer any kind of files, including VisiCalc (saving much redevelopment time), and use the package to transmit electronic mail between these previously incompatible computers, as well as between Apples and between PCs. The documentation is written for business people and professionals, we are told, and the package comes with a tape that interactively introduces the product. The program itself uses English commands and is menu-driven. $195. **Reader Service No. 125.**

Other Stuff

Store 80 5¼-inch floppies in the **Safer Floppy Disk Storage System** from MC^2. For $77 you get a lockable plastic filing cabinet (13.8 by 8 by 6.5 inches), suspension clips for a suspended filing system, dividers, reference cards, and switch-operated retaining bars. Disks are positioned with program information visible and the housing prevents dust contamination. **Reader Service No. 129.**

Get on-line information about microcomputer magazine articles and mini- and microcomputer software from **Dialog Information Services, Inc.** Search the Microcomputer Index, cross-referencing articles from thirty-five magazines about microcomputers, software and book reviews, applications, and new product descriptions (will my column be listed?, I wonder) for $45 per connect hour, and the International Software Directory for $60 per connect hour. Initially 10,000 records, it is updated monthly. Dialog's over 160 databases of over 60 million records include Compendex, abstracts from over 3500 engineering and scientific journals, conference reports, and government publications, and INSPEC, which they claim is the "largest English-language database covering physics, electrotechnology, computers, and control." **Reader Service No. 131.**

Remember the President's Analyst?

The phone company rather than cable companies stands to profit the most from the growth of home video services, *American Family* predicts in its monthly "Families and Telematics" column in the December 1982 issue.

The phone company will deliver most of the $38 billion in new services projected annually by 1990 – more than triple the value of entertainment services by then. This is because the phone company can do this more cheaply, more reliably, more privately, and with fewer restrictions than can cable, or so claims the phone company.

And He Even Looks a Bit Like Robbie the Robot

RB Robot Corporation makes the **RB5X** intelligent robot with an RS232 interface to connect to most micros for program entry and data transfer (at 1200 baud). RB5X learns from experience, has its own processor (National Semiconductor's INS 8073), memory (8K standard), programs (in Tiny BASIC), and tactile sensors. It senses when its own batteries (RB supplies rechargeable C and D batteries, four of each) are low, automatically finding its battery charger, recharging, and then resuming its peregrinations. Your basic stripped down RB5X costs $1195; you can add 16K more memory, a Polaroid Rangefinder sonar sensor, and a programmable flashing light for $295. Coming soon are a mechanical arm, voice synthesizer, and radio communications between robots. **Reader Service No. 127.**

Calls for Papers

Australian Computer Conference

Those among us who are contemplating defection to a more tranquil clime may wish to submit papers to the 10th Australian Computer Conference which is being held September 12-15, 1983, in Melbourne. Papers are solicited in the following major areas:

Stream I

The Trials, Tribulations and Triumphs of the Manufacture of Hardware and Software. Papers may be on financial, marketing, technical, managerial, research-development, educational or legal topics. Any other topic will be considered.

Stream II

The Application of Computers in Government, Commerce, Small Business, Education or Industry for all sizes of systems, as well as mixtures of micros and large machines. Topics include data base management, communications, graphics, and office automation. Any other topic will be considered.

Stream III

Managerial Aspects of Computing, including staff education, objective settings, audit and security, feasibility studies, staffing and motivating, time management, planning and controlling, documenting systems, estimating and budget setting, and social and legal responsibilities. Any other topic will be considered.

Stream IV

New Concepts in Computing: overview tutorials; data bases (including distributed DB); software components, system builders and portability; research/development presentations; fourth generation languages; communications/satellite/packet switching/local area networks; artificial intelligence; new computer architecture; information analysis; and telematics.

Stream V

Schools Symposium. Topics from every sphere of computing relevant to school students and teachers of computing.

Stream VI

The Development and Application of Graphic Systems: application of current technology to all spheres but particularly to the development of earth resources; and new developments and research into graphics software and hardware.

An abstract of the paper up to 1000 words must be submitted by January 1, 1983. All communications indicating the intention to submit a paper will be answered. A draft paper must be in by March 1, 1983. The final typescript paper must be submitted by June 14, 1983.

Address all correspondence to: Professor A. Y. Montgomery, 10 ACC, P.O. Box 4063, Mail Exchange, Melbourne, Victoria, 3001 Australia.

Association for Computational Linguistics

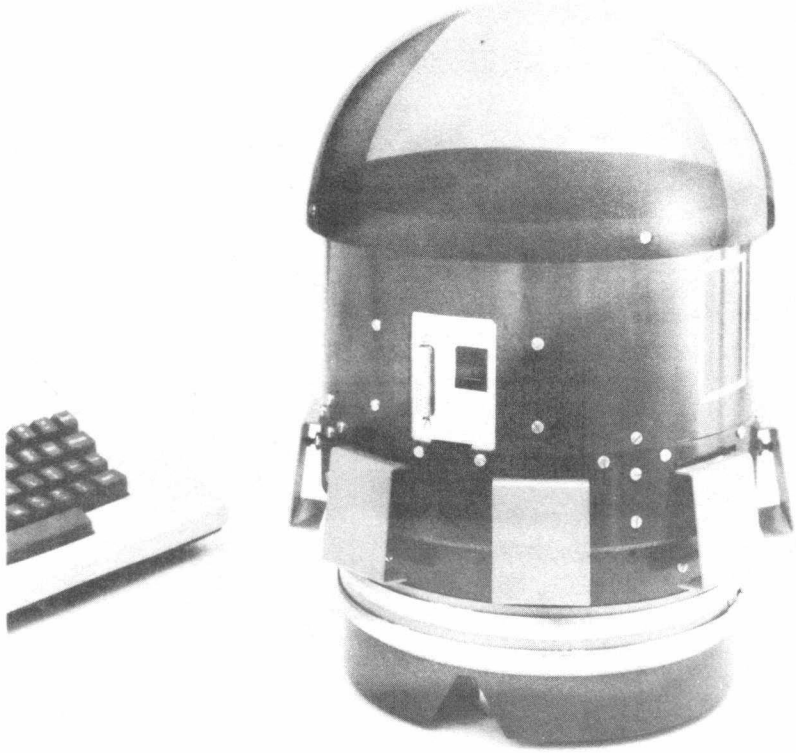
The 21st annual meeting of the Association for Computational Linguistics will be held June 15-17, 1983, at the Massachusetts Institute of Technology, Cambridge, Massachusetts.

Papers for the meeting are solicited on linguistically and computationally significant topics, including but not limited to the following:

- Syntax, parsing, and language generation, including aspects of linguistic theories relevant to computational models.
- Computational semantics, including logic, reference, anaphora, and metaphor.
- Discourse analysis and speech acts.
- Representation of knowledge, deduction, and planning as they relate to language understanding or production.
- Speech analysis and synthesis.
- Machine translation, machine-aided translation, and automated dictionaries.
- Formal foundations of computational linguistics.
- Applied computational linguistics.
- Software tools for computational linguistics.

Authors wishing to present a paper should submit six copies of a five- to eight-page summary, double-spaced, by February 1, 1983 to: Mitchell Marcus, 2D-443, Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974, USA; (201) 582-8948.

It is important that the summary contain sufficient information, including references to relevant literature, to convey the new ideas and allow the



program committee to determine the scope of the work. Authors should clearly indicate to what extent the work is complete and, if relevant, to what extent it has been implemented. A summary exceeding eight double-spaced pages in length may not receive the attention it deserves.

All submissions will be read by the program committee: Mitchell Marcus, Bell Laboratories, Chair; Philip Cohen, Fairchild Camera and Instrument Corporation; Lauri Karttunen, University of Texas, Austin; William Mann, USC/Information Sciences Institute; Robert Moore, SRI International; Ann Robinson, Symantec; Robert Wilensky, University of California, Berkeley.

Authors will be notified of the acceptance of their papers by March 7, 1983. Full length versions of accepted papers received by April 18, 1983 will be included in the Proceedings of the Conference.

Local arrangements are being handled by Jonathan Allen, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA 02139, USA; (617) 253-2509; ARPANET: ja@mit-ai.

For other information about the ACL, contact Don Walker (ACL), Artificial Intelligence Center, SRI International EJ278, Menlo Park, CA 94025, USA; (415) 859-3071; ARPANET: walker@sri-ai.

Goin' to the Shows?

The first quarter of 1983 promises a gaggle of computer shows. The Bay region will play host to three important microcomputer shows. In the first quarter, the International CP/M Conference, **CP/M 83**, will be held in San Francisco in January, followed by **Computer Swap America** in February in San Jose, and ending with **The West Coast Computer Faire** in San Francisco in March.

CP/M 83 is the first software show based on an operating system. CP/M, developed by Gary Kildall of *Digital* Research, has been the operating system of choice over the past seven years. The show is expected to draw OEM's, manufacturers, distributors and vendors from personal, business, office and scientific microcomputing sectors along with minicomputer and foreign companies. The show promoters, Northeast Expositions, originally offered 300 booths, but demand increased the size of the show to 650 booths. Admission is $20 and show dates are January 21st to the 23rd (Moscone Center, San Francisco).

Computer Swap America, which follows CP/M 83 by two weeks, is fast becoming an outlet for the entire computer industry (micro, mini and mainframe) to sell overstocked, surplus, obsolete, slow-moving, and many new products to computer hobbyists. The Bay Area has one of the largest concentrations of hobbyists and personal computer enthusiasts in the country. This no-frills, "good times & good deals," mini show has attracted manufacturers, retailers (computers and consumer electronics), software vendors and individuals from all over America to sell their wares. Admission is $5. The first of the swap shows in 1983 will be held on February 5th, from 10 to 6, at the Santa Clara County Fairgrounds in San Jose, California. Sellers should call (415) 494-6862 for further information.

The West Coast Computer Faire, which follows in March (18th-20th) has become an institution within personal computing and has grown right along with the industry. With the exception of Comdex, it has become the most successful pound-for-pound computer show of the year—a must for vendors and users alike. The 1983 Computer Faire will be held in Brooks Hall, San Francisco Civic Auditorium. Admission is $15.

Mini- and microcomputers, software, graphics, data and word processing, telecommunications, peripheral equipment, supplies and computer services—all will be found at the February 18th-21st **Second Annual Pacific Computer Expo** being held at the San Diego Convention and Performing arts center. There will be popular hands-on workshops conducted by Radio Shack, Apple Computer, Inc. and IBM Corporation, where one can sit down with a computer and evaluate its ability to meet your business, educational, home or personal needs. The Expo will feature some 150 exhibitors in 70,000 square feet of display area which will include computers, products and services from $10 to $125,000. Expanded seminar sessions will offer over 100 presentations covering all aspects of computer technology. Computer music, robots and a host of other electronic wonders for business, office and home will be at the Expo.

The **1983 Greater Baltimore Hamboree and Computerfest** will be held on March 27th, 1983 at the Maryland State Fairgrounds Exhibition Complex at Timonium, Maryland. A large exhibit area will display and sell the latest computers and software as well as the latest in electronics. Guest speakers will present a wide range of interesting topics throughout the day. The show will be open from 8 a.m. to 4 p.m. Admission is $3.00, children under 12 are free. Deluxe arrangements are available for dealers. Flea market spaces are available for hobbyists. For additional information, contact: GBH & C, P.O. Box 95, Timonium, Maryland 21093-0095; (301) 561-1282.

As you may have noticed, the Of Interest column is in a new format. We hope that its modularity makes it more readable. Dr. Dobb's *has a tradition of interaction with and among its readership; we hope that you will continue to respond. —Ed.*

DDJ

mounts of BASIC code to enable data entry and editing. MCDISPLAY allows the display to be defined by fields. The same display is used by the program operator for data editing as well as data entry. The programmer only needs to write one program instead of two or more. An entire "screen" of information is always in view by the program operator.

For further information, contact: Master Computing, Inc., P.O. Box 17442, Greenville, SC 29606; telephone (803) 244-8174.

Robot Triplicity at Atari

Atari, that game show host that greets you (even when you turn off your television set), that mega-company founded on PONG, the company that said it would never go to CP/M, is in fact doing just that. *Dr. Dobb's* has heard the rumor from three independent sources. Now, since putting three and three together gets you nine, the not so obvious conclusion is that Atari is probably building a robot or robot-like product which will probably run on CP/M. Their simulations for games like Battle Zone are one possible connection too, since these "real world" terrain emulating matrices are derived from actual military technology. This robot will probably be self-charging, like its competition, and will be programmable to perform certain household functions such as smoke sensing, burglar or intrusion detection, and remote telephone stationing, all with speech synthesis, if desired. We can't wait to have our very own micro-mouse "Bot" droid for under two grand, but will it go to the 7-11 for "munchies" on a rainy night? Only time will tell!

Atari's entry into the "Bot" market may turn out to be a moot point since a Colorado-based company called the RB Robot Corporation is a full year ahead with a magical little "druid" (droid) called RB5X*. This guy is about the size of a wastebasket, runs on Tiny BASIC (see *DDJ* number one) and is available now. No matter how rich they are, the brains at Atari will have to play catch-up poker, at least for the first year. The RB5X will be growing ahead of Atari, offering add-ons, arms, sensor packs, lasers, etc. Interestingly enough, Heath has also announced its entry into Robotics and a Heath-Kit Robot will soon be available. These are the first companies producing actual home-available "Bots" for under $2000, but the market is infinitely expandable.

I sure wish we had one!

See page 70 of DDJ's Of Interest, this issue, for a picture of the RB5X. –Ed.

Peddling In Nice

Chuck Peddle, the developer of the Commadore PET, the 6502, the VIC, and the VICTOR 9000, has hinted that the European headquarters of his newly formed Victor Computer Company (which he claims will be the third largest computer company in the world someday) will not be in Paris or its environs, but rather in the more pleasant and decidedly nude spot on the Riviera called "Nice." Why not, he just announced an improved Victor 9000 with 10 Mb for under $6000. That should put IBM on notice. There is no doubt that the new Victor-Sirius system is a bargain, but . . . "Gee, Chuck, will we really ever get any work done on the Cote d'Azur?"

Very Rair Black Box

Besides the Victor 9000, a British company stole the show at Comdex. This gutsy little company, which licenses a PC to ICL, this IBM of England (although it is not owned by ICL as some believe), took a stand in the middle and did very well indeed.

The Rair Black Box is at least five years old. ICL is new to the micro world and chose Rair to develop their new PC, a beige, fluffy thing that looks like a marshmallow. The Black Box, however, looks like a Porsche and runs as well. Even without screen and keyboard, it is superior to any but the Victor 9000 in this price range. This machine can be compared to any 16-bit system, and this includes the overly-honored IBM PC.

The 8-bit Black Box is companioned by the Rair line, plus a new 16-bit Black Box still in production. In 1981, I had some hands-on with the system in Ireland and loved it because it had a 5Mb integral hard disk, great fun for those days and ready to go at a time when American companies were tinkering with the idea. Dynabyte was the first to do anything about it, but that project evolved into their current "biggie" called the Monarch.

The Black Box hasn't changed much since its introduction. It is built to EEC standards which take into consideration longevity and operational normalcy in hostile environments. Like the microprocessor-based Pontiac on television (the one that talks to the driver), the Rair Black Box looks tough and runs smooth, but when you punch in STAT *.* and it returns the number, 14.650K, to the screen, you know you're on to something. The hard disk has grown from 5Mb to 16Mb optional, although the 5Mb is still available. The hard disks can be chained in increments of 5, 10 or 16Mb. The model we saw at Comdex was a 32Mb version with over 500K of RAM. There is also a Rair business computer and Rair does its own R&D and production.

Dr. Dobb's is currently a beta installation for the Rair equipment and the good Doctor is reviewing the system for next month's issue. The 350 and 360 top-of-the-line Black Boxes use an 8085 with memory management that can address up to 1Mb of RAM.

The Rair system does not arrive with a keyboard or terminal, but it was shown at Comdex with the ADDS keyboard and terminal as featured by Bove and Rhodes in the premiere issue of *The User's Guide*. Rair Black Boxes feature an advanced implementation MP/M II.

For further information, contact: The Rair Computer Corporation, 4101 Burton Drive, Santa Clara, CA. Their telephone number is (408) 988-1790.

DDJ

This tends to send you to a tight loop in never-ever land.

Until Digital Research makes BDOS calls reentrant, do not try to trace through them. Your best bet is to break point before and after the call and examine the registers. I hope this explains the problem and will prevent someone from learning about it like me (painfully).

Stephen M. Kenton
Software Consultant
475 College
Norman, OK 73069

A Letter From the Editor

Dear Readers:

We are putting together an advanced S-100 system. We have all of the components. If there are any hardware jockies in the San Francisco Bay Area who would be interested in this project, drop us a line. A small honorarium and lots of free treasure await.

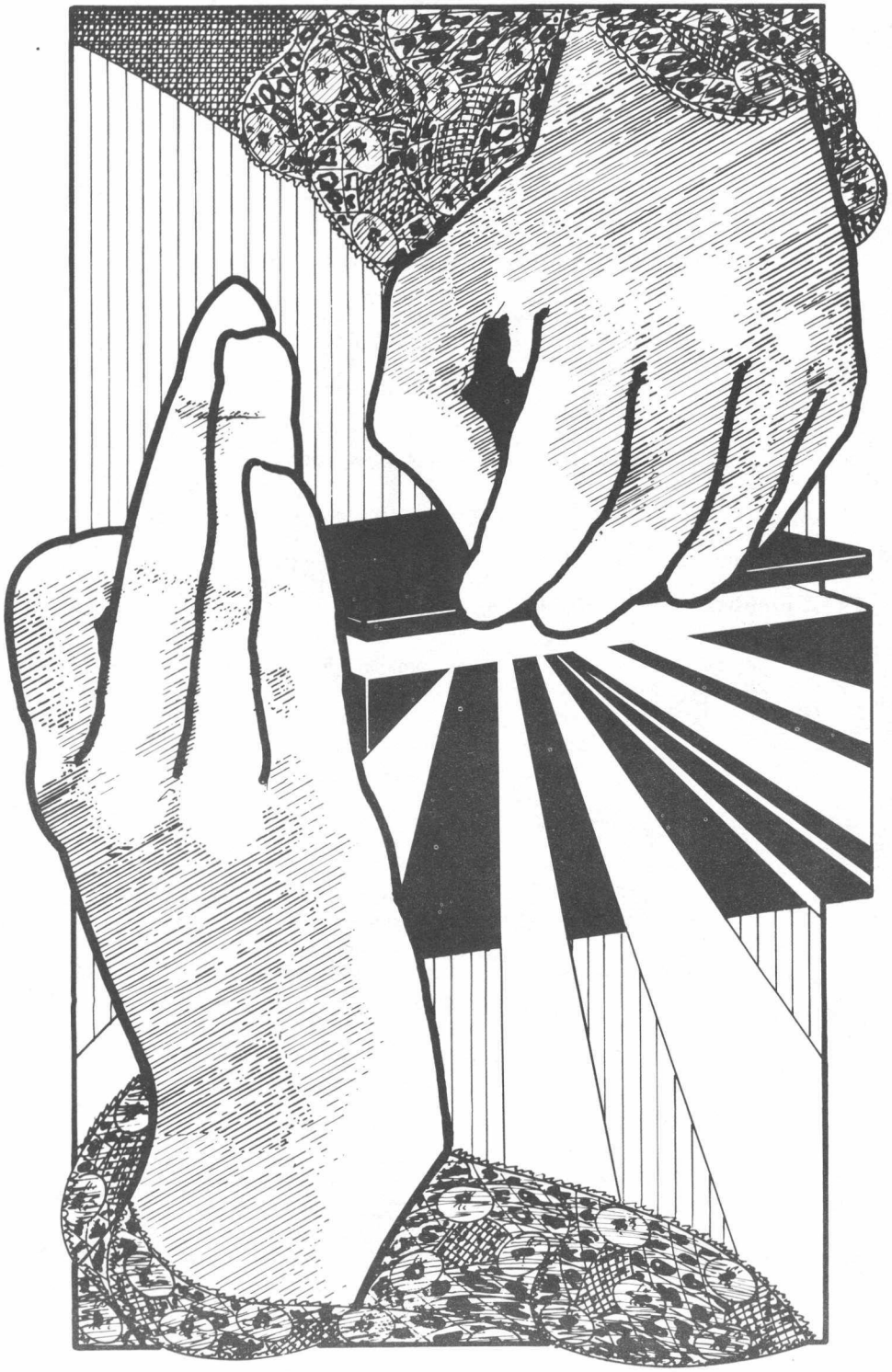
Hank Harrison, Editor

DDJ

#77 March 1983 $2.50 ($3.00 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems



AUGUSTA Part II

A Small-C Operating System

6809 Threaded-Code

Faster Smaller BASIC

For Users of Small Computer Systems

March 1983 Volume 8, Issue 3

Publisher/Art Director – Clifford West
Senior Editor – Hank Harrison
Associate Editor – Reynold Wiggins
Contributing Editors –
Dave Caulkins, Dave Cortesi, Ray Duncan, Gene Head, Marlin Ouverson, Michael Wiesenberg
Marketing Director – Craig Harper
Marketing Manager – Beatrice Blatteis
Advertising Director – Carl Landau
Ad Sales Representative – Doug Millison
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Terri Marty, Linda Marohn
Production Manager – Barbara Ruzgerian
Production Assistant – Jane A. McKean
Typesetter – Paula Fairchild
Cover Illustration – Al McCahon

© 1982 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus, Paul Lutus, Ben Goldfarb, Rodney Black, Steven Weiss, Pat Phelan, John Brodie, James Fleming, Unison World, Thomas Davis, Jan Steinman, Ronald E. Johnson, G. Hunter, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III, Infoworld, Stan Veit, Western Material Control, S.P. Kennedy. **Sustaining Subscribers:** Pete Roberts, Steven Fisher Enterprises. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

CONTENTS

20 Augusta, Part II – The Augusta P-Code Interpreter
by Edward M. Mitchell
In the January issue, Mr. Mitchell described the structure of a subset of Ada called Augusta. This month he looks at the p-codes and at the p-code interpreter of Augusta.

36 A Small-C Operating System
by Brian McKeon
Adding a new twist to our continuing coverage of the development of Small-C, author McKeon has provided for your experimentation, the source code for an operating system developed with Ron Cain's Small-C compiler.

62 6809 Threaded Code: Parametrization and Transfer of Control
by H. T. Gordon
The author discusses how careful optimization of Forth-like languages, taking into consideration the superlative addressing power of the newer CPUs, can result in insignificant reduction of timing penalties. He argues that major conceptual changes such as parametrization will streamline languages and provide some common core structure to them.

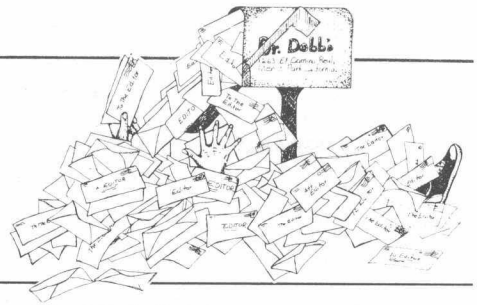
66 A Common-Sense Guide to Faster, Smaller BASIC
by Robert Irving
Speedy CPUs and lots of RAM are great, but are your coding habits creating memory bloat and an idling processor? These down-to-earth guidelines could help fat cats and bare boards alike to program more efficiently.

71 A Fundamental Mistake in Compiler Design
by Edgar H. Fey, Jr.
The huge size of recent assemblers and compilers prompted Mr. Fey to provide us with his observations on the reason for the bloat, and some suggestions for the reduction of it.

6 Letters
14 16-Bit Software Toolbox
Cromemco's dual CPU, more on memory tests, an item regarding 8086/88 segment registers, and IBM low-resolution graphics revisited.
75 Open Forum
78 CP/M Exchange
The first of a two-part column on disk I/O.
82 Dr. Dobb's Clinic
A calendar algorithm, a slick way to use user numbers, symbol generation, and more on the IBM PC – cursor movement, and the mechanical *translation of IBM* BASIC.
86 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

LETTERS



Off to a Good Start

Dear Editor,

Your January issue was stimulating as usual. Of the dozen or so computer journals that arrive every month, *Dr. Dobb's* is one of the few that I always read. The good Dr. Dobb deserves particular thanks for introducing me to the Forth language several years ago.

On the whole I disagree strongly with Tom Pittman's January letter. But he does have a valid point with regard to those interminable listings. Everybody has a modem these days. Perhaps you should consider putting the program files on Compuserve or The Source or an in-house dial-up line. Even at 300 baud, it sure beats typing.

I enjoyed Ray Duncan's discussion of the numeric evaluation of trigonometric functions. As usual, he is brief and cogent, but on one point his trigonometry is a little bit rusty. We do not have to extract a square root to obtain the cosine. The cosine is simply the sine shifted by 90 degrees. Thus cos(x)=sin(x+90) or for radian phreaks, cos(x)=sin(x+PI/2).

To the entire staff at *Dr. Dobb's*, I wish a successful and prosperous new year. Keep up the good work.

Sincerely,
Joseph McDermott
265 Bogert Road
River Edge, NJ 07661

A Slightly Slicker Fix for Small-C

Dear Editor,

Mr. Macpherson was kind enough to send me a copy of his letter to you (*DDJ* #76, pp. 10-11) describing a Small-C bug involving *continue* statements within *switch* statements. He was quick to fathom the problem and find a solution. After studying his patch, I settled on a slightly more efficient and straightforward solution. I hope he won't mind if I recommend this patch over his own.

First, in the function *doswitch* below the call to *addwhile* (line 184, page 39, *DDJ* #74), insert the line

```
*wqptr + WQLOOP - WQSIZ) = 0;
```

Then change the following functions to read as shown. Modified lines begin with a # character.

(page 42, *DDJ* #74)

```
    dobreak() {
     int *ptr;
#   if ((ptr=readwhile(wqptr))==0)
            return;
    modstk ((ptr[WQSP]), NO);
    jump(ptr[WQEXIT]);
    }

    docont() {
     int *ptr;
#   ptr = wqptr;
#   while (1) {
#    if ((ptr=readwhile(ptr))==0) return;
#    if (ptr[WQLOOP]) break;
#    }
    modstk((ptr[WQSP]), NO);
    jump(ptr[WQLOOP]);
    }
```

(page 45, *DDJ* #74)

```
    delwhile() {
#    if (wqptr > wq)
            wqptr=wqptr-WQSIZ;
     }

#   readwhile(ptr) int *ptr;{
#    if (ptr <= wq) {
#     error("out of context");
      return 0;
     }
#    else return (ptr-WQSIZ);
    }
```

This will compile with the original compiler, so those bringing it up for the first time with this patch should have no problem.

The response to my article has been most encouraging. I was surprised to find that a number of well known microcomputer companies are using the little compiler. I am looking forward to seeing more software contributions written for Small-C.

Sincerely,
J. E. Hendrix
Rt. 1, Box 74-B-1
Oxford, MS 38655

Augusta Short Circuits

Dear Dr. Dobb,

I was very pleased to see the article, "Augusta," by Edward Mitchell. Ada™ is unfortunately "doomed to succeed," so it is good to see that those of us who live on small machines will be able to see all the reasons why it is being panned by such notables as Hoare ("The Emperor's Old Clothes," *Communications of the ACM*, Vol. 24, No. 2, February 1981). I will also be interested in seeing a compiler written in BASIC, of all things. Unfortunately, Mr. Mitchell left out one of the most useful features of Ada, the package.

However, Mr. Mitchell states that the "short circuit" conditionals of Ada are unusual. I beg to differ. Many modern languages, such as C and LISP, support short circuit evaluation of conditionals. Most languages (including, notably, Pascal) leave the question of whether conditionals short circuit up to the implementor. In fact, one of the popular Fortran compilers for the IBM 370 would test Mr. Mitchell's example [if (n<>0) and (j/n=1) then] by testing whether or not j/n=1, and if that were false, "short circuit" the test n<>0. This state of affairs means that the defensive (i.e., smart) programmer will act as if things will *not* be short circuited, and also not put statements with side effects in conditionals, just in case things *are* short circuited. After working with such things, it is truly enjoyable to use something which lets you know what's going on – whether it be C, LISP, or Ada.

Mike Meyer
Box 1749
Norman, OK 73070

Compliments for Kossow

Dear Dr. Dobb,

Thanks for presenting Allen Kossow's 68000 cross-assembler. Mr. Kossow has done an excellent job of managing the large instruction set and numerous addressing modes of this new processor.

In the past I have made use of system software listings provided by the doctor. This time, however, because of the imminent arrival of a 68000-based development system at my workplace, I took advantage of Mr. Kossow's offer. Twenty-five dollars is a reasonable copying fee in my opinion.

I was pleased to find that the single-density RT-11 diskette contained an executable file in addition to the Fortran and Macro source files. Because of this, I immediately set to testing. Five problems were discovered using examples from *68000 Assembly Language Programming* by Kane, Hawkins and Leventhal (Osborne/McGraw Hill, 1981).

The first hitch was an illegal instruction trap encountered when run on a PDP-11/04. It was traced to the Macro routine file, specifically, the ASH instruction in routine GETBIT. Replacing this one instruction with four (4) ASR R0 instructions cleared things up nicely. If you are using a DEC processor with EIS

(Extended Instruction Set) like an LSI-11/23 or LIS-11/34, then this change is unnecessary.

The second problem showed up as a reversal of the source and destination registers in all ADDX, SUBX, ABCD, and SBCD instructions. A fix is as follows:

Replace these three lines in subroutine PRCESS

```
1910 IF(OP1EA.EQ.5)
       OPSKEL=OPSKEL+8
     OPSKEL=OPSKEL+OP2DA
     OBJBUF(1)=
      OPSKEL+(OP1DA*"1000)
```

by inserting these three lines

```
     OPSKEL=OPSKEL+8
1910 OPSKEL=OPSKEL+OP1DA
     OPJBUF(1)=OPSKEL+
      (OP2DA*"1000)
```

The third bug prevented any MOVE instruction from being assembled as MOVEQ(quick move). A fix follows:

Replace these four lines in subroutine PRCESS

```
     IF(IMODE.NE.3)GOTO 304
     IF(OPNWRD(3).EQ.0) GOTO
      301
     IF(OPNWRD(3).EQ.-1) GOTO
      301
     GOTO 304
301 . . .
```

by inserting the following three lines

```
     IF(IMODE.NE.0) GOTO 304
     IF(OPNWRD(3).EQ.0) GOTO
      301
     IF(OPNWRD(3).NE.-1) GOTO
      304
301 . . .
```

This causes the assembler to optimize a MOVE immediate instruction to the MOVEQ form only if the datum size is defaulted and the datum is in the legal range for MOVEQ(−129 < arg < 128). For a consistent treatment of quick mode instructions, the following change should be made to make ADDQ and SUBQ operate in the same manner:

insert this line into PRCESS

```
     IF(IMODE.NE.0) GOTO 536
```

in between these two existing lines

```
     IF(OPNFLG.EQ.1) GOTO 536
```

and

```
     IF(OPNWRD(2).GE.1.AND.
      OPNWRD(2).LE.8) GOTO 550
536 . . .
```

Problem number four prevented assembly of the CMPM form of the CMP instruction. A fix is below:

Replace this line in subroutine PRCESS

```
400 . . .
     IF((OP1EA.EQ.5) .AND.
      (OP2EA.EQ.5)) GOTO 480
```

by this new line

```
     IF((OP1EA.EQ.4) .AND.
      (OP2EA.EQ.4)) GOTO 480
```

A fifth problem was noted in assembly of the LEA instruction where the addressing mode is register indirect with index. An example is LEA -1(A0,D0.W), A1. I have not had an opportunity to analyze the problem any deeper.

My compliments to Allen Kossow. A piece of software of this size seldom appears with so few glaring errors. Keep up the good work.

Sincerely,
Steve Albrecht
Tracor Northern Instruments
Middleton, WI 53562

Dear Editor,

I have an ulterior motive for writing. I *must* have one of Al Kossow's (*DDJ* No. 68, "Multi-68000s in a Personal System") computers.

I've written to Mr. Kossow to express my enthusiasm. A project like his is an ambitious one so I didn't really expect an answer.

Yes, I know he said any future developments would be published in *DDJ*. The point of *this* letter is to urge *DDJ* to publish the info . . .

Anxiously awaiting each and every issue . . .

Judd Ellmers
1A Rolling Ridge Road
Montvale, NJ 07645

Editor's note: We hope to do an update soon. In the meanwhile, can any DDJ *readers help us out on multi-68000s?*

More on Graphics Algorithms . . .

Gentlemen:

The vector generation algorithm published in the December issue of *DDJ* is not new. Several years ago, I purchased a Houston Instrument plotter and the manual included an implementation in BASIC of what was essentially that algorithm. I found the algorithm described in one of the BYTE Books called *Bits and Pieces*. There it was credited to somebody at IBM. Most implementations reduce the calculation per step even further than Mr. Michalski's version. Usually F is initialized to 2*DY-DX and in the loop, F is tested relative to zero as the first step of the loop. If F is negative, only X is incremented before the new point is plotted and F=F+2*DY is calculated. If F is zero or positive, both X and Y are incremented before the new point is plotted and F=F+2*DY−2*DX is calculated beforehand so only one addition per step is necessary to update F. Here I have used the same notation as Mr. Michalski except for using D instead of delta. Well, it is a good idea no matter how many times it is reinvented.

This is the simplest and probably the most useful of what I call the where-do-we-go-from-here algorithms. The idea is that the choice of the next point to be moved to or set is limited to one of the eight points next to the last point. This limitation may be inherent in the graphics device or imposed to insure the drawing of a continuous line. Knowledge of the general direction of the curve is used to limit the choice to two of these points, and some easily calculated error function is used to choose between the two. So far this straight line algorithm and the one for doing circles and arcs are the only ones for which I have worked out the details and successfully implemented. I did both of them in BASIC first and then finding this a bit slow, I resorted to a mixture of Fortran and assembly language. With that I was able to keep my plotter going at close to its maximum speed. Although I have not yet worked out the details, it appears that these techniques could be extended to any of the conic sections, but at a price of increasing complexity and decreasing speed.

Another graphics related matter that I have been planning to write you about is related to Mr. Taylor's ellipsoid problem which appeared in the Clinic. I was one of those that sent in a solution to his problem. Ever since his response to the solutions appeared in the August issue, I have been thinking about it and I have reached the conclusion that he posed the wrong problem. He talks of perspective projection and yet the problem that he posed was projection parallel to the Z axis which is orthographic projection. I have worked out the solution for perspective projection, but since it involves even more calculation than the solution for orthographic projection, which he found to involve too much calculation for his purposes, it probably will be of little use to him.

Sincerely yours,
David S. Tilton
27 Pennacook Street
Manchester, NH 03104

. . . And Variations on Michalski's

People:

I was interested in your simple vector generation algorithm in the December *DDJ* – interested because I had devised the same underlying algorithm for use in a mailing list application.

A businessman has a mailing list with 90 names (for example). He wishes to send some promotional material, but to only 20 people (small budget, maybe). Therefore he wants a way of extracting 20 names from the list of 90 in an even distribution.

Hopefully, you can see that selecting 20 names from 90 is a similar problem to

drawing a vector which rises 20x points and runs 90y points.

The algorithm used in the mailing list application was: Let y run from 0 to 89, and set x=y*20div90. X will increase slowly from 0 to 19. For each input name (y), print it out only if the value of x has increased from the previous y (integer arithmetic). In Pascal:

```
var
   x, y, xprev : integer;
begin
   xprev := -1;
   for y := 0 to 90-1 do begin
      getname;
      x := y * 20 div 90;
      if x <> xprev then
         putname;
      xprev := x
   end
end
```

I hope this shows the idea. Obviously a useful program would use variables in place of 90 and 20, and probably would have to make an initial pass through the list to count how many names are there. The flowchart published in the article uses more code but is more efficient because it eliminates the multiply and divide operations.

An amusing variation: The above program tries to print 20 names, but if the original list only had 8 names, it would only print 8. By replacing the "if" statement with a "for" loop, you can guarantee 20 printed names no matter how small the mailing list is (provided it's not empty).

```
. . .
getname;
x := y * 20 div 8;
for xprev := xprev to x-1 do
      putname;
xprev := x
. . .
```

With this program, each of the 8 names is printed two or three times to give an output list of 20 entries.

Yours truly,
Peter Raynham
10 Camrose Crescent
Scarborough, Ontario
Canada M1L 2B6

Forth Cosine with 8087

Dear Editor:

The purpose of this letter is twofold. (1) I wish to share with your readers my enthusiasm toward Forth Inc.'s method of structuring 8087 floating point in their Forth for the IBM PC. (2) Since trigonometric functions are not included in Polyforth (at least not in the version I own), I would like to put into public domain a program (see Listing 1, this page) for calculating cosines. Besides being fast (470 microseconds) and accurate (16 digits), the algorithm I developed is unusual because it uses an unspecified 8087 feature and involves no conditional branching. Thus, the algorithm should be of interest to non-Forth programmers.

Forth is not a popular floating-point language and has no floating-point standards. Traditionally, floating-point numbers have been stored with integers on the parameter stack which resides in memory. This is somewhat inefficient since there are no operations between the two numerical types and one tends to obstruct the other, thereby requiring extensive stack manipulation in a floating-point program.

Consider the 8087, a chip with a stack of its own comprised of eight 80-bit registers that can perform most internal operations faster than a 64-bit number can be transferred to memory. Indeed, the 8087 can swap two 80-bit numbers on its stack faster than the 8088 can swap two 16-bit numbers in memory. Clearly, with the 8087, it would be a serious waste of time to automatically transfer 8-byte floating-point numbers to a memory stack and then back again between operations.

Forth Inc. has made a rational deviation from tradition by simply using the 8087 stack as an extra Forth stack. It wouldn't surprise me if this chip-language combination is unbeatable among microcomputers for speed. To test my hypothesis, I invite benchmark challenges, particularly from owners of 68000-based computers. I only require that the program be a practical 64-bit number cruncher such as matrix multiplication and not an abstraction.

Regarding the cosine algorithm, from the angle, V, the 8087 instruction FPTAN returns the sides of a right triangle, X(V) and Y(V). Intel literature states the restriction, "0<V<PI/4"; however, I discovered that my 8087 was completely accurate in the range -PI/2<V<PI/2. Using this feature, I came up with the following algorithm that is best described in equation form:

```
Let U = Angle
S1*[FABS( FABS[ FPREM
( U,2*PI ) ] - PI ) - PI/2]
          ( U,2*PI ) ] - PI ) - PI/2 ]
COS(U) = Y(V) / FSQRT( X(V)*X(V)
          + Y(V)*Y(V) )
```

Notice that multiplying by S1 keeps V from exceeding its limits. I have included an implementation of this algorithm using Polyforth assembler mnemonics. The sine function can be obtained by simply subtracting PI/2 from U.

Sincerely yours,
Steven A. Ruzinsky
2110 S. Austin Blvd.
Cicero, IL 60650

(See Listing 1 below)

[See correction in April issue, page 8. -- Ed.]

Subscription Glitch

Dear Doctor Dobb,

I'm sure you have heard stories like mine many times before, but I would be grateful if you eyeballed my output.

I used to be a small microprocessor with very underdeveloped software. When I went to the beach, big strong mainframes would kick sand in my face and attractive young minis would laugh at me.

How delighted was I when I discovered what a dose of *DDJ* did for my development! So wonderful were the results, that it is little surprise that I soon became addicted to monthly shots of *DDJ*. Naturally when my subscription renewal became due, I raced down to the bank at my fastest baud rate, obtained a bank cheque in U.S. dollars for two years (yes, I wasn't going to risk cold turkey), and sent it off airmail immediately.

Can you imagine my alarm when some months later another renewal notice

Listing 1.

8087 Cosine Routine in Polyforth.

```
254
 0 ( 8087 Cosine in Polyforth, 470 microsec. )
 1
 2         LVARIABLE S1
 3         .9999999999999999 S1 L!
 4
 5 CODE
 6  COS   FLD1 FLDPI FSCALE 1 N) FSTP 1 FXCH
 7        FPREM 1 N) FSTP FABS FLD1 FCHS FLDPI
 8        2 N) FSUB -POP FSCALE 1 N) FSTP
 9        1 FXCH FABS 1 N) FSUB REV R64 S1 FMUL
10        FPTAN 1 N) FLD 0 N) FMUL NO 1 FXCH
11        0 N) FMUL NO 1 N) FADD FSQRT 1 N) FDIV
12        NEXT
13
14
```

arrived! Of course I straightaway photocopied the bank cheque duplicate and sent it off with an impassioned plea.

Now (fear and trembling) I have been told my subscription has expired because the cheque was not received. Bless my chips, what will I do? I know it cannot be a computer error because we never err. To make matters worse, my bank tells me it will be several weeks before they can find out whether the bank cheque was banked at the other end.

Please, Doctor Dobbs, help a poor deprived little processor whose last shot of *DDJ* was November 1982! Please scan your files to make sure whether my cheque has arrived. If my bank tells me the cheque is lost, I will send another as fast as I can.

Yours faithfully,
(signed for Neil's computer by)

Dr. Neil Trezise
Wild Dog Creek Road
St. Andrews
Victoria 9761
Australia

Ed. Note: Those of you who have experienced similar problems, please bear with us. During our recent change of data services, a number of our sheep became separated from the fold. We are working hard to get this corrected, but unfortunately these things take a bit of time. Sorry for the inconvenience. Just let us know (most of you already have) and we will make the correction.

Standard Deviation

Hi guys,

Good to see that C compiler is progressing; nothing better than a good free software. I'd like to say a few things about C pertaining to portability (a very important issue).

I've been seeing more and more C code that does not follow the guidelines published by Kernighan & Ritchie (*The C Programming Language*). That's a real no-no. If everyone adheres to those guidelines with care, C will remain a very portable language. J. E. Hendrix is guilty of the crime himself in his December article. Since publication of the compiler implementation and its libraries is a restatement of the standard, it should be done with careful attention to detail.

Hendrix describes several functions in his auxiliary library which do not maintain the spirit of the K & R teachings (the C bible and de facto definition). His dtoi, decimal to integer, conversion function sounds almost the same as K & R's atoi (ASCII to integer). Doesn't matter which name sounds better – atoi is the standard. Also atoi only requires one parameter; dtoi unnecessarily requires two (as do Hendrix's utoi and xtoi). Also discordant with K & R's atoi function, Mr. Hendrix's itou and itox also require one more parameter than they should. The string size parameter is useful but, probably more often than not, not necessary. Perhaps these functions should be retained, given names to denote their special ability, and not published as the main functions of their type in a "standard" library.

I consider the loose interpretation of the C standard as it pertains to its libraries a major problem for its future portability. I've tried several of the compilers available commercially and found them all to have problems in this area.

Both the BDS-C and the Supersoft C compilers have non-standard fopen functions. That's one function they should know enough to leave alone. BDS doesn't implement the mode feature at all, and requires a pointer to a buffer be passed to it (why?). Supersoft simply adds an additional argument allowing you to specify your buffer size. Handy perhaps, but non-standard! Why not make it an additional function by a different name if you need the "enhancement." Ah, and what about the Cadillac – Whitesmith's? It too requires an additional argument to specify that the buffer address be passed to its fopen. But that's just the beginning for Whitesmith's.

Their enhancements (read "incompatibilities") are legion. Believe it or not, they don't even have a printf function! Or a scanf. They have putfmt and getfmt instead. They are slightly more flexible, and don't use the old standard names which is good, but why no printf and scanf? Whitesmith's list of small "enhancements" goes on and on: strcpy is called cpystr, strcmp is called cmpstr, atoi is called btoi (buffer to int), itoa is called itob, etc. My pessimistic side says they do this to lock you in to their family of C compilers (8080, PDP-11, 68000, etc.).

Come on guys, it's a great language; don't muck it up.

Anyone even halfway serious about using C and perhaps building his own extended library (that's what it's all about) should read the K & R book from cover to cover at least twice, and pay close attention to the style the fathers of C have developed and to their many good examples. Their's is only one way of doing it. But it's a good way, and it's the standard.

Not to detract from the very important C problem, I'd like to make a note in reference to the "Interrupts and CP/M" article. Circular or ring buffers, as they're called, are very natural with most processors, and very handy as the article shows. I think Mr. Bromberger may have missed the point that many things in the computer world are naturally circular. As an example, his code to keep the buffer pointer within the limits of the buffer:

```
        inr
        cpi
        jc
        mvi
int1:
```

can easily be replaced with the following code which is smaller and, I think, more indicative of the buffer's circularity.

```
        inr   a
        ani   7fh
int1:
```

This requires that the buffer size be an even power of two, and the mask (7fh in the example) be one less than the buffer size.

I'd also like to mention that I feel that little flap with the JRT people is a good reaffirmation of *DDJ's* separation of editorial and advertising policies. Too bad for JRT, they really did us all a service by starting the $29.95 movement.

Thanks for a great magazine, and to these authors in particular for some very accessible and useful information.

Sincerely,
Richard Foulk
Pegasus Software
P.O. Box 10J
Honolulu, HI 96816

What's the Holdup?

Dear Editor:

JRT Pascal seems to be a current topic of interest among owners of micro computers. There have been several letters published reporting problems with the $29.95 Pascal system. Maybe this will be a new activity to be engaged in. I am sure that the problems that the owners have found with JRT Pascal are real. If JRT Systems is not going to solve these problems, I feel certain that the readers of *Dr. Dobb's Journal* will soon be reporting their solutions. Since the JRT System is priced within the reach of every computer owner, there will be many working on meeting the new challenge.

I have a problem that is not new with those of us who order by mail. I had my order processed on September 22, 1982 by JRT Systems. I have not received my JRT Pascal as yet. Several letters to JRT Systems have gone unanswered. I hope that my program arrives soon, and then I can start finding if JRT Pascal is good or BAD.

Sincerely,
Donald M. Dealy
231 Washington St.
S. Attleboro, MA 02703

Ed Note: The preceding letter was not the first that we have received of its kind.

(Continued on page 90)

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

Cromemco's Dual CPU

One of the most pleasant fringe benefits of starting up a software house is that it gives one the opportunity to buy all sorts of beautiful new hardware for "Research and Development." When Cromemco announced their combination Z-80 and 68000 "DPU" board early this year, I was fascinated but wary. The board seemed as though it potentially could give the same impetus to the 68000 among personal computer users as the Godbout Dual CPU board gave the 8088: providing a safe, known development environment to fall back on (Z-80 and CP/M-80), while giving ready access to the power of the 68000 microprocessor. However, I was a bit skeptical because most of Cromemco's boards in the past have not completely followed the S-100 bus standard, and were difficult to mix and match with components produced by other vendors. In addition, Cromemco's software is expensive and incompatible with standard CP/M application packages. Of course, being a computer junkie, I ordered the board anyway.

About three months after I put down a deposit, the board finally arrived and immediately got stuck on the shelf for another two months due to the pressure of other work. Finally, during the December lull, I found time to drag the DPU out and look it over. The component is built on a typical S-100, silk-screened, multilayer printed circuit board, and displays the usual solid Cromemco design and construction. It contains about sixty integrated circuits including the familiar looking Z-80 and the monstrous looking 64-pin 68000. In a departure from Cromemco's previous practice, all components except the two microprocessors and three other support ICs are wave soldered directly to the board instead of being mounted on sockets. The board is designated as "Revision E" in the silk-screened legend, but it still has no less than eight little green wires snaking their way around – this leads one to believe that the design of this board is not too stable yet.

The DPU board automatically starts up in the Z-80 mode of operation. When a 1 is output to port 0FFH, the DPU switches to the 68000 mode. The first time the 68000 is used, it obtains its stack pointer from location 0 and its program counter from location 4. To give control back to the Z-80, the 68000 must write a zero to location 0FFFFFFH (its I/O is memory mapped). Subsequently, when either processor resumes execution, the contents of its registers and instruction pointer are unchanged from the previous invocation. Since there is no direct communciation between the registers of the two microprocessors, information must be passed back and forth by temporarily saving it in memory. Listing 1 shows how to switch between the two microprocessors under software control.

I can guess what you're thinking if you've read this far. In these times of economic unease, few of us can afford to junk our 8-bit systems and shell out several thousand dollars for a complete complement of Cromemco boards including memory and disk controller in order to get started with a workable 68000 system. But I have good news for you. I was tipped off by a friend that the Dual CPU board was "more S-100 compatible" than previous Cromemco products. In a moment of rashness, I set the jump-on-restart address for the Z-80 to 0000 by cutting two jumpers, dropped the DPU board into my Compu-Pro Z-80 system, and turned on the power. The system booted up and ran perfectly!

This opens up enormous possibilities for experimentation with the 68000. You can do all of your software development with a familiar CP/M-80 support structure for disk and terminal I/O. When CP/M-68K becomes available, you can write your initial BIOS and cold boot loader under CP/M-80, just as many of you did when the Godbout 8085/8088 CPU board was first released.

By the way, you will find *68000 Assembly Language Programming* by Gerry Kane *et al.* to be an extremely useful reference work as you are getting started. This book is devoted entirely to describing the instruction set from a programmer's point of view and is filled with helpful examples; it does not waste any space describing the hardware – for this you can refer to the manufacturer's literature.

I should note in passing that Compu-Pro has also started delivering their 68000 CPU. It costs only $625 in contrast to the $1000 for the Cromemco board; however, it doesn't have the advantage of the on-board Z-80. Compu-Pro is also selling a stand-alone 68000 Forth system for $200, but is not yet delivering the Digital Research operating system.

More on Memory Tests

Some readers will recall that when I reviewed the Microsoft RAMCARD for the IBM Personal Computer, I had some critical things to say about the memory test program that came with it. At the time, I was just a bit disgusted with the program's style of interaction with the operator but had no reason to find fault with the actual usefulness of the program.

I must report that the program has a much more serious defect than I imagined: if the board it is testing has read errors causing parity errors, the test program crashes completely! Instead of getting the dandy display of the error type and location as the manual specifies, the parity

Listing 1.

Switching between the Z-80 and the 68000 on Cromemco's DPU board.

```
; switching on the 68000
;
;
          ld        hl,init     ;move initial values
          ld        de,0        ;for stack pointer and
          ld        bc,8        ;program counter into
          ldir                  ;locations 0 and 4
                                ;
          ld        a,1         ;now turn on the 68000
          out       0ffh,a      ;


; switching on the Z-80
; (i/o ports are memory mapped in the top)
; (64 k of the address space on the 68000)
;
          move.b    #0,0ffffffh
;
```

error causes a hardware interrupt forcing transfer to the ROM BIOS, the message "PARITY ERROR2" is displayed on the screen, and the system goes dead. Apparently the test program does not reset the interrupt transfer vector so that it can intercept these events and perform the appropriate error analysis.

Regarding 8086/88 Segment Registers

The Intel 8086/88 documentation is rather vague about the use of default segment registers in the various indirect addressing modes. The *iAPX 86,88 User's Manual* states on page 2-12 that "most variables (memory operands) are assumed to reside in the current data segment, although a program can instruct the Bus Interface Unit to access a variable in one of the other currently addressable segments. . . ."

Leo Scanlon, the author of a book on 8088 assembly-language programming (which is in press at the Robert J. Brady Co.), has written to point out a surprising exception to this rule. The base indexed addressing modes [BP+SI] or [BP+DI], which I had always assumed were used to extract data from multi-dimensional data arrays, do *not* use DS (Data Segment register) as the default but rather form their address in combination with the SS (Stack Segment register). Teleologically, this is a bit unexpected, but Leo proved it experimentally with the program in Listing 2. Table 1 is derived from a similar table in Leo's new book and is reprinted with his permission as an aid to *DDJ* readers.

IBM Low Resolution Graphics Revisited

In an earlier column I printed a BASIC listing which demonstrated how to initialize and use the low resolution color graphics mode on the IBM Personal Computer, allowing the display of sixteen color pixels with a resolution of 160 by 100. James Murphy, of Madison, Wisconsin, has kindly sent in a set of Forth routines to plot points in the lo-res mode (see Listing 3); these execute about twenty times faster than the original BASIC version even though they are written in high-level code.

In Screen #75 of the listing, the definitions "TR" and "CRT_LRC_INIT" take care of setting up the video controller chip's various registers. "CLEAR_BUFF" initializes the video memory map with the required values so that each pixel will be displayed as a solid block of color. "LRCG" (mnemonic for Low Resolution Color Graphics) uses the previously defined words to initialize the display and prepare for plotting. In Screen #76, "!DOT" (Store Dot) accepts an address and a color and sets the appropriate pixel in the memory map. X must be in the range 0-159 and Y in the range 0-99, with (X,Y)=(0,0) being the upper left corner of the screen. Finally, Screen #77 contains a nice little demonstration program which draws a test pattern of nested squares in different colors.

Readers who wish to extend the low resolution graphics capability even further can refer to the Forth line drawing routine which was printed in the July 1982 issue of *DDJ*.

DDJ

Addressing Mode	Operand Format	Segment Register
Register	reg	none
Immediate	data	none
Direct	disp	DS
	label	DS
Register indirect	[BX]	DS
	[BP]	SS
	[DI]	DS*
	[SI]	DS
Base relative	[BX+disp]	DS
	[BP+disp]	SS
Direct indexed	[DI+disp]	DS
	[SI+disp]	DS
Base indexed	[BX+SI+disp]	DS
	[BX+DI+disp]	DS
	[BP+SI+disp]	SS
	[BP+DI+disp]	SS

*In string instructions, the destination address is formed by the combination of DI and ES, and the ES register cannot be over-overridden.

Notes:
1. "disp" is optional for base indexed addressing.
2. "reg" can be any 8- or 16-bit register, except IP.
3. "data" can be an 8- or 16-bit constant value.
4. "disp" can be an 8- or 16-bit signed displacement.

Table 1.
8088 addressing modes and segment register defaults.

(Listing 3 on page 18)

Listing 2.
Leo Scanlon's program to determine the default segment registers for base indexed addressing on the Intel 8086/88.

```
STACK   SEGMENT   PARA STACK 'STACK'
        DB        100 DUP(0FFH)
STACK   ENDS
DATA    SEGMENT   PARA 'DATA'
        DB        0ABH

DATA    ENDS
CODE    SEGMENT   PARA 'CODE'
PROG    PROC      FAR

        ASSUME    CS:CODE,DS:DATA,SS:STACK
        MOV       AX,DATA   ;initialize DS
        MOV       DS,AX
        SUB       BP,BP     ;clear the
        SUB       SI,SI     ;addressing regs.
        SUB       DI,DI
        MOV       DL,[BP+SI] ;read memory with
        MOV       DH,[BP+DI] ;based indexed addr.
        RET

PROG    ENDP
CODE    ENDS
        END       CODE
```

Listing 3.
Laboratory Microsystems PC/Forth.

```
Screen # 75
  0 ( Lo Res Color Graphics                                    12/27/82 )
  1 FORTH DEFINITIONS HEX
  2
  3 ( initialize video controller chip )
  4 : TR              3D4 PC! 3D5 PC! ;
  5 : CRT_LRC_INIT  9 3D8 PC! 07F 4 TR 64 6 TR 70 7 TR 1 9 TR ;
  6
  7 ( initialize video memory map for low res graphics )
  8 : CLEAR_BUFF     4000 0 DO ODE B800 I !L  2 +LOOP ;
  9
 10 ( set operating mode for low resolution color graphics )
 11 : LRCG           2 MODE  CRT_LRC_INIT  CLEAR_BUFF ;
 12 -->
 13
 14
 15


Screen # 76
  0 ( Lo Res Color Graphics                                    12/27/82 )
  1
  2 ( plot point in low res graphics:  x y color --- )
  3 : !DOT            ROT DUP 1 AND
  4                   IF    ROT 0A0 * + B800 SWAP 2DUP C@L
  5                         0F0 AND >R ROT R> OR ROT ROT C!L
  6                   ELSE  1 OR SWAP 10 * SWAP ROT 0A0 * +
  7                         B800 SWAP 2DUP C@L 0F AND
  8                         >R ROT R> OR ROT ROT C!L
  9                   THEN ;
 10
 11 DECIMAL
 12 -->
 13
 14
 15


Screen # 77
  0 ( Lo Res Color Graphics demonstration                      12/27/82 )
  1 3 VARIABLE COLOR
  2 0 VARIABLE CSEQ     2 , 4 , 6 , 8 , 10 , 12 , 14 ,
  3                   1 , 3 , 5 , 7 , 9 , 11 , 13 , 15 ,
  4 : SQUARE          DUP 80 + OVER 80 SWAP -
  5                   3 PICK DUP 50 + 50 ROT - SWAP 1+ SWAP
  6                   DO    DUP I COLOR @ !DOT OVER I COLOR @ !DOT
  7                   LOOP  2DROP DUP 50 + OVER 50 SWAP -
  8                   3 PICK DUP 80 + 80 ROT - SWAP 1+ SWAP
  9                   DO    DUP I SWAP COLOR @ !DOT
 10                         OVER I SWAP COLOR @ !DOT
 11                   LOOP  2DROP DROP ;
 12 : DEMO            LRCG 50 0
 13                   DO    I 15 AND 2* CSEQ + @ COLOR ! I SQUARE
 14                   LOOP  KEY DROP 3 MODE ;
 15
```

AUGUSTA, Part II

The Augusta P-Code Interpreter

by Edward Mitchell

Part 1 of this four-part series described the Augusta language. This month's article continues with a description of the p-codes and the p-code interpreter that executes Augusta programs.

Stack Machines

The Augusta compiler translates source statements into code for a hypothetical "pseudo-machine" based on the use of stacks. Stacks provide a natural representation for arithmetic expressions and for implementing nested procedure calls and local data definitions.

Owners of Reverse Polish Notation (RPN) calculators are already familiar with stacks and their use for evaluating expressions. The basic operations performed on the stack are push and pop, to add and to remove data to and from the stack, respectively.

A stack is often compared to a spring operated stack of plates in a cafeteria. A new plate is added by *pushing* it on to the top. When a plate is removed, the whole stack *pops* upwards. For example, if we push both 13 and 10, the stack looks like

```
10
13
```

Then, we can add them together to give

```
23
```

If we translated these steps to an assembly-like language, we might write

```
PUSH 13
PUSH 10
ADD
```

Complicated expressions like 5*(3+7) are coded as

```
PUSH 5
PUSH 3
PUSH 7
ADD       ; Add 3+7
MULT      ; Multiply result by 5
```

With a few more instructions, the stack becomes a general purpose expression evaluator. For example, to perform the assignment, X:=10+13 on a stack machine, the following sequence of instructions might be executed

```
PUSH address of X
          ; Put the memory address
          ; of X on stack
PUSH 10
PUSH 13
ADD       ; Compute 10+12,
          ; leaving 23 on stack
STORE     ; Store top of stack value
          ; into the address con-
          ; tained in the next word
          ; on the stack
```

The Augusta p-machine consists of about 80 instructions (see Table 1) to manipulate the stack and perform other functions such as conditional testing, branching, procedure calls and returns.

P-Machine Overview

The p-machine is a 16-bit stack-oriented hypothetical computer. It has a code pointer register CP that contains the byte address of the next instruction to execute, and a stack pointer SP that points to the top of the stack. In addition, the global frame pointer GF and the local frame pointer LF are used to speed up references to global and local variables, respectively.

Registers SB (Stack Base) and CB (Code Base) are of use when debugging programs and point to the bottom of the stack, and to the first byte of code for the currently executing procedure. CS (Code Segment) points to the address of the first byte of code. The number of the procedure that is executing is stored in the PN register (the first "PROCEDURE" seen in a program becomes procedure 1, the second "PROCEDURE" statement becomes procedure 2, and so on).

Before a program can be run, it must be loaded into memory. One more register, PT, contains the address of the procedure table. The procedure table contains a 7-byte entry for each procedure in the program and includes (1) the address in memory of the procedure's code, (2) the

Corrections to Augusta, Part I

Several last minute editing changes were made by both the author and the editors at *Dr. Dobb's* in order to shorten the length of Part 1. Unfortunately, a few incorrect and misleading statements crept into the text.

The Augusta compiler is written in Microsoft BASIC, and hence, can be run on systems other than a standard CP/M system. However, compiled programs can only be executed by using the p-code interpreter, a program that currently runs only on Z-80-based CP/M systems. While I do hope to produce an MS-DOS version, it was premature for the editor's note to announce that such a version will be available shortly.

Originally, Part 2 of the series described a BASIC language implementation of the p-code interpreter. But that description was removed and some of the comments in Part 1 no longer apply. For example, the BASIC p-code interpreter implemented virtual memory but the assembly-coded version of the interpreter does not.

The variable declaration

```
S : STRING;
```

defines S as a string of up to 80 bytes (not 128, as shown in Part 1). A different length is specified by writing

```
S : STRING(200);
```

which gives S a length of 200 bytes. Maximum string length is 254 bytes.

Part 1, Table 1 summarized only a few of the available "built-in" procedures and functions. With nearly 40 procedures, Augusta provides an Ada-like file system, printer and serial port I/O, string searching, MOVELEFT, MOVERIGHT, and a very limited dynamic memory allocation scheme.

To support file access, a fifth predefined type is used, as

```
F : IN_OUT_FILE;
```

which defines F as a file access variable. File variables may be grouped in arrays and passed as OUT procedure parameters.

The syntax diagrams contained three minor errors. Corrections are shown in Figure 1 (opposite). Procedures and functions are not required to have parameter lists as was implied by the original diagrams, and an ELSE may not follow an ELSEIF.

Copyright © 1983 by Edward Mitchell, Graduate, Computer and Information Sciences, University of California, Irvine.

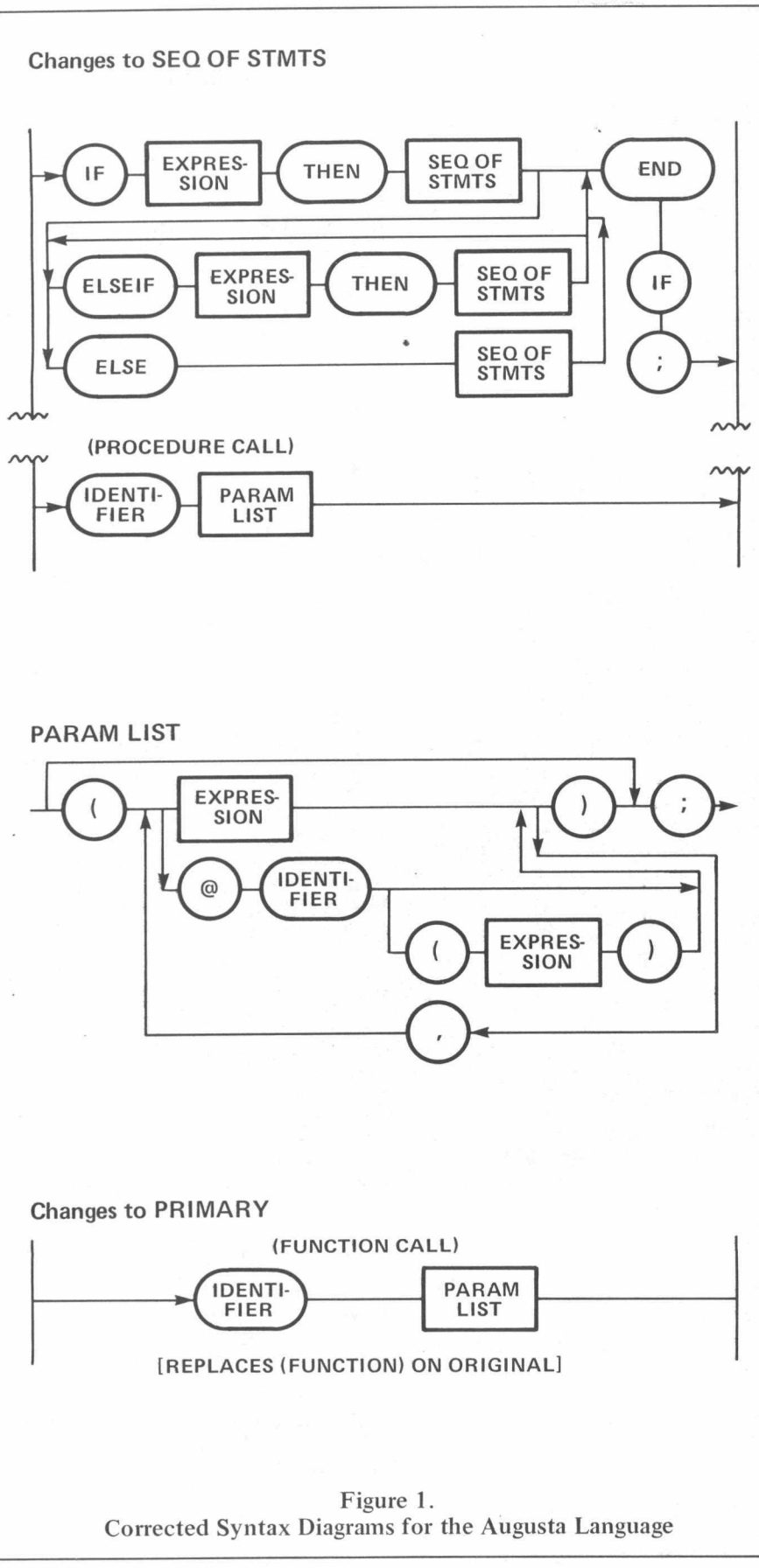


Figure 1.
Corrected Syntax Diagrams for the Augusta Language

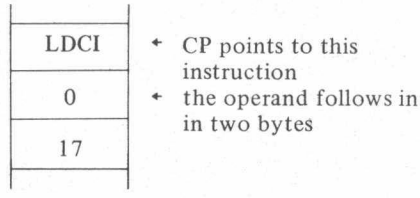
number of bytes of parameter variables, (3) the amount of memory required for local variables, and (4) the "lexical level" of the procedure.

All procedure calls are made through the procedure table. The instruction CGP 5, for example, means to call procedure 5. The location of procedure 5 and how much memory to allocate for its local variables are fetched from entry 5 in the procedure table.

The P-Code Instructions

The entire group of instructions is divided into several smaller groups consisting of (1) "load" and "store" instructions, (2) string assignments, (3) logical operations, (4) integer arithmetic operations, (5) integer comparisons, (6) string comparisons, (7) jump instructions, and (8) procedure calls (see Table 1, pages 31-33).

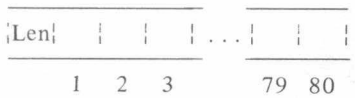
A simple operation is to load a constant on to the stack. The LDCI or "Load Constant Integer" opcode is a single byte long, followed by a one-word (16-bit) value. When the p-code interpreter is executing instructions, CP always points to the next byte of the program code. For LDCI, the p-machine reads the instruction and decodes the opcode byte. Then it jumps to the appropriate LDCI subroutine to load the constant. The actual constant is stored in the two bytes following the instruction, as shown here:



After the LDCI opcode is read, CP is incremented to point to the first byte of the constant. The LDCI subroutine then reads the 16-bit integer and pushes it on the stack. A push is performed by writing the word into the memory location pointed to by SP, and then incrementing SP by two (since one word occupies two bytes and SP always points to a word). CP is incremented so that it points to just past the second byte of the constant, which is the next instruction.

Four data types are recognized by the p-machine: (1) 8-bit bytes, (2) 16-bit integers, (3) fixed-length strings, and (4) boolean values. An integer takes two bytes and represents numbers in the range -32768 to 32767. A string uses a fixed amount of memory determined from its declaration in the Augusta program, plus one byte to hold the current length of the string. For example, if S is a string of up to 80 characters in length, then 81 bytes

of storage are reserved. The first byte holds the length as illustrated here:



For example, the string "Alphabet" would be stored in memory as

A boolean TRUE is any non-zero word while FALSE is a word containing zero.

A variety of instructions are used to load the values or addresses of both global and local variables. Several variations on these instructions are provided to access "intermediate level" variables (these occur when procedures are defined inside of procedures that are defined inside of procedures, ad infinitum) and special "short" forms that reduce the size of the compiled programs.

A single instruction "STO" writes the word on the top of the stack into the address contained in the second word from the top of the stack.

Strings are never loaded on to the stack. Instead, just a pointer to the string is loaded. For example, when the statement

```
S := "STRING";
```

is compiled, Augusta translates this to a "Load Constant Address" (LCA) instruction, followed by the string assignment operator SAS. The constant string "STRING" is translated by Augusta into

```
LCA 6 S T R I N G
      ↑
Length Byte
```

When the p-machine encounters the LCA opcode, it pushes the address of the length byte on to the stack. Then it sets CP to point to the first byte after "STRING."

Logical and arithmetic instructions perform an operation to the top one or two words of the stack. For example, ADI pops the top two words from the stack, adds them together and then pushes the result. Relational operators, like EQUI, pop the top words and compare the values. If the relation is true, then a non-zero word is pushed, otherwise a zero is placed on the stack.

Stacks and Procedure Calls

When a program executes a procedure call, the computer uses a stack to keep track of the "return address." Each time that a procedure is called, the return address is pushed. If a called procedure, in turn, calls another procedure, then both return addresses are distinct. The return address of the most recently called procedure is always the topmost address on the stack.

When a procedure finishes, it pops its return address from the stack and resumes execution at its calling point. As we'll see in a moment, the stack is also used to hold additional information. The portion of the stack that holds the return address is known as the "frame mark." The frame mark delineates each procedure from the other procedures on the stack and saves the state of the procedure that was running before the procedure call.

Variable Allocations

Augusta permits *nested* procedure definitions. Each procedure can have its own *local* data. Space for local variables is allocated when the procedure is called. When the procedure exits, all of the space used by the local variables is made free.

Figure 2 (page 24) shows how a stack machine allocates storage space for variables defined in the outermost procedure and for local variables defined within called procedures. The variables in the outermost procedure are accessible to all of the procedures that are contained within it and are referred to as "global" variables. Global storage space appears on the bottom of the stack.

When a procedure is called, any needed parameters are pushed, a frame mark containing the return address and other information is output, and the stack pointer is incremented high enough to leave space above the frame mark for local variables. When the procedure finishes executing, the information from the frame mark is restored, and the stack pointer is cut back to where it was before the parameters were pushed. In this manner, all local storage space is allocated dynamically at the time of the procedure call, and then disappears upon return to the caller.

Global variables are referenced as an offset from the global frame mark at the base of the stack. For example, a variable at offset 2 is referenced with the "Load Global Value" instruction

```
LDO 2     ; Load Global instruction
```

The location of the global frame is kept in register GF. To fetch the value at offset 2, the p-machine adds 2 to GF which computes the memory address of the global memory word.

Local variables are referenced as an offset from the local frame mark so that the instruction

```
LDL 3     ; Load Local instruction
```

references the data three bytes above the local frame mark. Register LF always contains the address of the currently executing frame.

Parameter variables are also referenced with the local data instructions. Since parameters are located *below* the frame mark, the local instruction uses a negative offset. For example,

```
LDL -16   ; Load the value of
          ; first parameter
```

Handling Nested Procedures

Augusta, like Ada and Pascal, allows nested procedure definitions as in Figure 3 (page 28). Procedure P4, inside P3, can reference variables in both P3 and P2. These variables are neither "global" nor "local" to P4. Instead, they are "up-level" references.

After P4 has been called, the stack is as shown in Figure 3(b). To reference variable Y in procedure P3, the p-machine has to refer to the previous frame on the stack. Similarly, to reach variable X, the p-machine must traverse backwards two frame marks to reach the local data of procedure P2. Because of up-level references, the frame mark needs to contain more than just the return address.

The return address or "dynamic link" chains together the frame marks in the order that the procedures were called. If procedure P4 makes a recursive call on procedure P3, P4 *cannot* be the previous frame for an up-level variable reference. If it were, then P3 could up-level address variables for a lower level procedure. Therefore, the dynamic link is insufficient for up-level addressing. Instead, a *static link* is used to reflect the structure of the program.

When P3 calls P4, the static link of P4's frame mark points to the frame mark of P3. If P4 calls P3, a new frame is created for P3 on top of the stack. However, P3's static link is set to point to the frame for P2, because P3 is declared inside of P2.

In addition to the dynamic and the static links, the frame mark also contains the number of bytes of parameters, the lexical nesting level, and the code base pointer of the current procedure. The lexical level is the depth of nested procedure definitions. If P2 is inside of P1, the first procedure in the program, then P2 is at lexical level 2. Similarly, if P3 is defined inside of P2, then P3 is at lexical level 3.

Code File and Memory Layout

The p-code interpreter is contained in a code file named RUN. (Note: An alternate p-code *interpreter* named DRUN contains a built-in debug utility. The utility provides program tracing, single-step and *n*-step execution, and examining and changing the value of variables.) The interpreter reads the code file into memory and begins execution.

The code file is organized into three sections (see Figure 4(a), page 29): (1) a 128-byte header block containing information about the code file, (2) a 1792-

byte procedure table containing space for 256 procedures (entry 256 is not used), and (3) the p-code for each of the procedures in the program.

When the code file is loaded, only the used portion of the procedure table is read into memory. Obviously, if the program only contains ten procedures, then only the first part of the table needs to be read.

After the code is loaded, the CP register is initialized to the first p-code in procedure 1, and the stack pointer is set to point to the first word after the p-codes. The other registers, including GF, LF, and PT, are set as needed. Figure 4(d) shows the memory layout after a code file has been read into memory.

Augusta Efficiency

One way to measure a language's efficiency is to run a benchmark test and compare the size of the generated code and its execution time to that of other languages. Listing 1 (page 35) is the source code of a simple prime number generator based on the sieve of Eratosthenes. (Listing 2, also on page 35, is the disassembly of that same program's object code into p-codes.) Table 2 (page 34) compares that program, written in Augusta, to versions written for several other well-known compilers (see references 1 and 2, below).

In the May Issue

The Augusta compiler is a one-pass recursive descent compiler. As the parser recognizes a language construct, it generates the appropriate code. Part 3 provides a general description of recursive-descent compilers.

(Figure 3 on page 28)
(Figure 4 on page 29)
(Figure 5 on page 30)
(Table 1 on page 31)
(Table 2 on page 34)
(Listings 1 and 2 on page 35)

DDJ

References for Part 2.

[1] Gilbreath, Jim. "A High-Level Language Benchmark," *BYTE*, September 1981, p. 180.

[2] Gilbreath, Jim and Gary Gilbreath. "Eratosthenes Revisited," *BYTE*, January 1983, p. 283.

(Figure 2a)

```
PROCEDURE P1 IS
  I : INTEGER;
  J : INTEGER;


PROCEDURE P2 ( X : INTEGER ) IS
  A : INTEGER;
  B : INTEGER;


BEGIN
  .
  .
  .
END;


BEGIN
  P2(3);
END;
```

(Figure 2b)

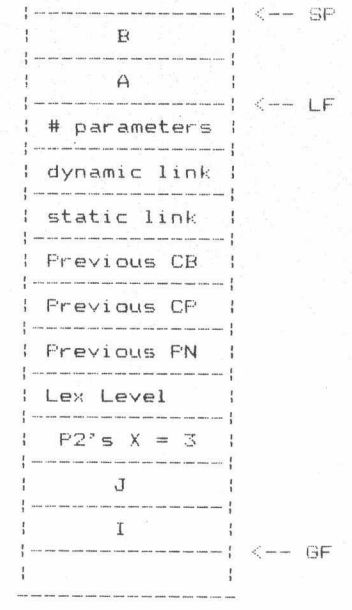


Figure 2.

Storage for variables is allocated on the stack. (b) is a map of the stack for the program shown at (a), after P2 was called. Space for I and J is allocated at the base of the stack. When P2 is called, its single parameter, 3, is pushed on the stack and topped with the frame mark for P2 and P2's local variables.

(Figure 3a)

```
PROCEDURE P1 IS
  W : INTEGER;

  PROCEDURE P2 IS
    X : INTEGER;

    PROCEDURE P3 IS
      Y : INTEGER;

      PROCEDURE P4 IS
        Z : INTEGER;
        BEGIN
          Z := Y;    -- ACCESS Y IN P3
          Z := X;    -- ACCESS X IN P2
        END; -- P4

      BEGIN
        Y:=1;
        P4;
      END; -- P3

    BEGIN
      X:=2;
      P3;
    END; -- P2

  BEGIN
    P2;
  END; -- P1
```

(Figure 3b)

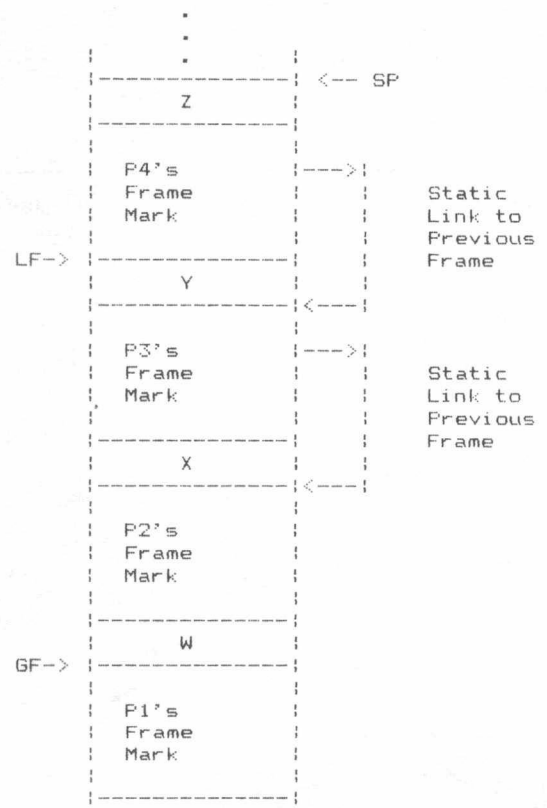


Figure 3.

Procedures can be defined inside of other procedures, as shown at (a). When P4 references variable Y, the p-machine must traverse down the stack to the previous frame mark to reach Y. Similarly, a reference to X is a reference to two frames lower in the stack.

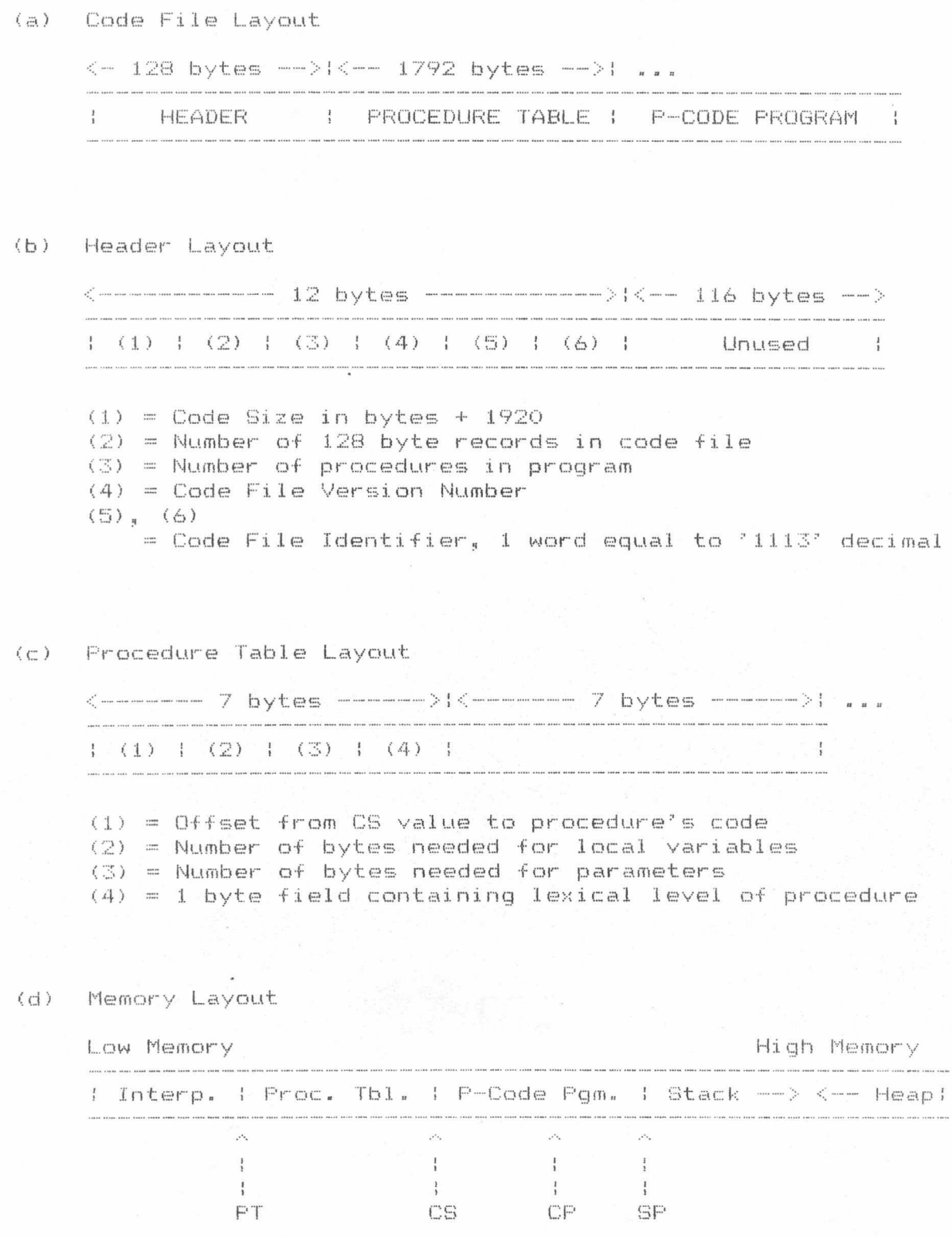


Figure 4.

Detailed layout of an Augusta code file and of memory after a program has been loaded. (a) shows the code file format, (b) and (c) detail the format of the header block and of the proceedure table, and (d) shows the memory layout of a loaded program.

(Continued on next page)

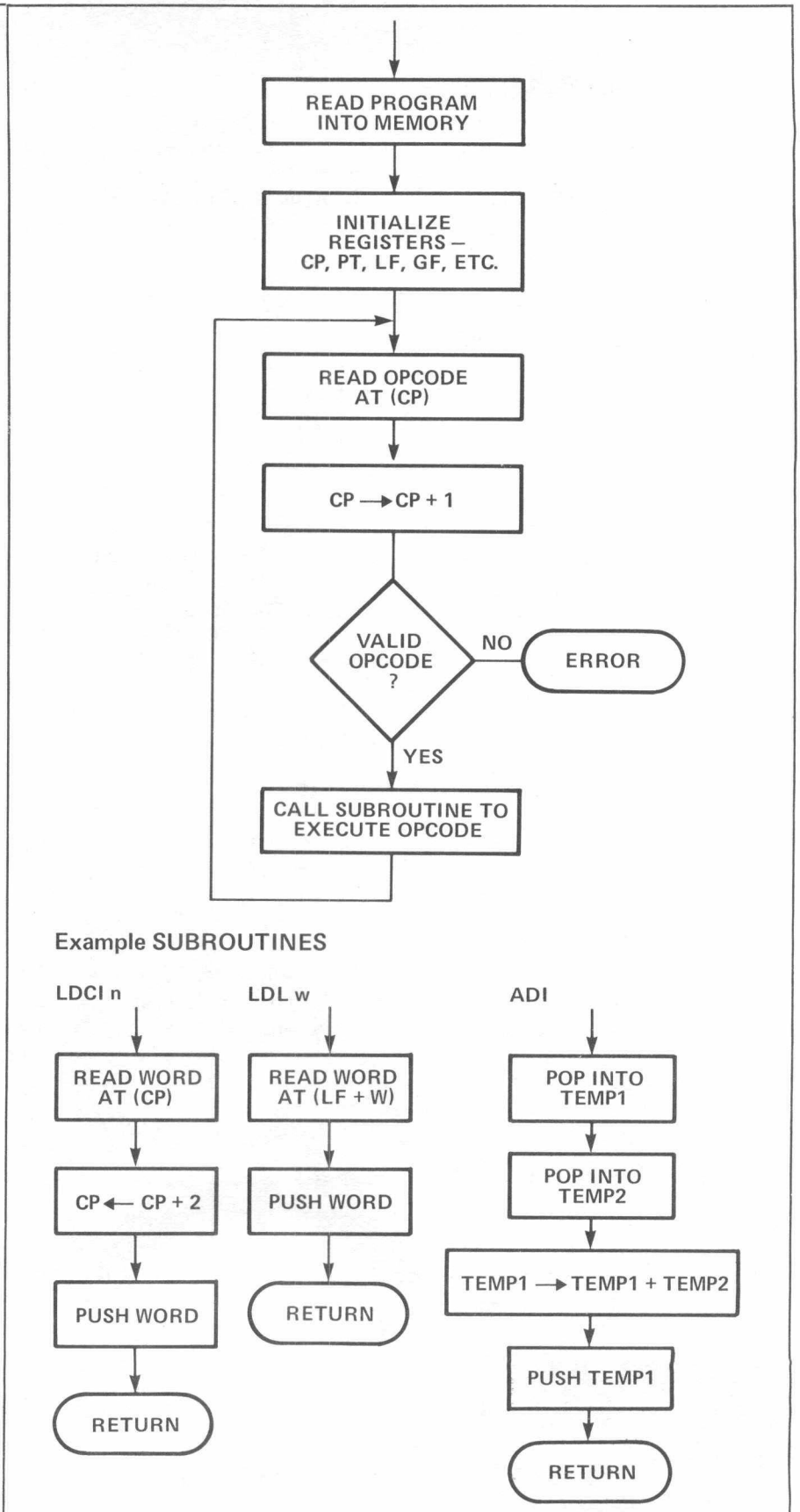


Figure 5.
Outline of Interpreter Logic

Table 1.

The Augusta P-Code Instruction Set.

All of the Augusta p-code instructions are shown in Table 1. The number in parenthesis is the opcode value. A "b" following the opcode indicates that there is a one-byte parameter. "w," "w1," and "w2" are 16-bit words. In the descriptions, TOS is the word on the top of stack, and TOS-1 is the word just below the top of stack.

```
Loads and Stores

( 1) LDCI w     - Load Constant Immediate Word

( 2) LDL w      - Load Local word at offset w

( 3) LLA w      - Load address of Local word at offset w

( 4) LDB        - Load the byte pointed to by the pointer in TOS.

( 5) LDO w      - Load global word at offset w

( 6) LAO w      - Load global address at offset w

( 8) LOD b, w   - Load word up level b static links at offset w

( 9) LDA b, w   - Load address up level, as above.

(11) STO        - Store indirect TOS into address in TOS-1

(12) SINDO      - Load indirect using the address in TOS

Short Loads

(57) SLDO b     - Short Load Global data at offset b.  b is in
                  the range 0..255.

(58) SLAO b     - Short Load address at global offset b.   b is in
                  the range 0..255.

(59) SLLA b     - Short Load Local Address at local offset b.

(60) SLDL b     - Short Load Local data at local offset b.

(61) SLDC b     - Short Load Constant in the range 0..255.

(49) to (56) SLDL0 .. SLDL7
                - Single byte opcodes to Short Load Local Data at
                  local offset 0 through 7.

(63) SLDCN1     - Short load negative 1 (-1)

(64) to (79) SLDC0 .. SLDC15
                - Single byte opcodes to short load a constant in
                  the range 0 to 15.

Array Indexing Instructions
(24) IND        - Index an integer array.  TOS-1 is the address
                  of the base of the array and TOS is an index
                  into the array.  IND pops TOS, multiplies it
                  by 2 and adds it to TOS-1, pushing the result.
```

(Continued on page 32)

(Table 1 continued from page 31)

```
(48) IXA b
                - Index array.  This is a generalized IND opcode,
                  where b substitutes for the "2" multiplier.

String Assignments
(13) LCA b, <Chars>
                - Load address of a string constant.  b is the
                  number of bytes in the string address.  <Chars>
                  are the characters in the constant.  Pushes the
                  address of the byte containing b and increments
                  CP to point to the next instruction following
                  <Chars>.

(14) SAS        - String Assignment.  TOS is a pointer to source
                  string  and TOS-1 points to a destination.  Copies
                  bytes from the source string to the destination string.

Logical Operators
(16) AND        - Boolean AND of TOS and TOS-1

(17) OR         - Boolean OR of TOS and TOS-1

(18) NOT        - Boolean complement.  TOS <-- NOT TOS

Integer Arithmetic Operations

(19) ADI        - Add the integers in TOS and TOS-1.

(20) NGI        - Let TOS equal -TOS.

(21) SBI        - Subtract integers.  TOS-1 minus TOS.

(22) MPI        - Multiply integers in TOS and TOS-1.

(23) DVI        - Divide integers, TOS-1 divided by TOS.

(45) MODI       - TOS-1 MOD TOS

(80) INCL w     - Increment word at local offset w.

(81) DECL w     - Decrement word at local offset w.

Integer Comparisons
                The comparions push -1 if the result is true, or 0
                if the comparison is false.

(25) EQUI       - Compare TOS to TOS-1 for equality. (=)

(26) NEQI       - TOS <> TOS-1

(27) LEQI       - TOS-1 <= TOS

(28) LESI       - TOS-1 <  TOS

(29) GEQI       - TOS-1 >= TOS
```

```
(30) GTRI       - TOS-1 >  TOS


String Comparisons
                TOS and TOS-1 are pointers to the strings
                to compare.  Pushes -1 if the comparison is true
                or 0 if the comparison is false.

(31) EQUSTR     - TOS-1 =  TOS

(32) NEQSTR     - TOS-1 <> TOS

(33) LEQSTR     - TOS-1 <= TOS

(34) LESSTR     - TOS-1 <  TOS

(35) GEQSTR     - TOS-1 >= TOS

(36) GTRSTR     - TOS-1 >  TOS

Jump Instructions
                All jumps are made relative to the location of the
                instruction.

(37) UJP w      - Unconditional jump to w bytes away from
                  current instruction.

(38) FJP w      - False Jump.  Jump if TOS is 0.

(39) XJP w1, w2, w3
                 - Case jump instruction.  See text for description.


Procedure Calls and Returns

(40) CLP b      - Call local procedure b.

(41) CGP b      - Call global procedure b.


(42) CSP b      - Call system procedure b.  System procedures
                  handle all input/output.

(46) CIP b      - Call intermediate level procedure b.

(43) RET        - Return from a procedure.

(46) RNP        - Return from a function by saving the TOS value
                  setting SP back before the parameters and frame
                  mark for the current procedure and then restoring
                  the TOS value.

Miscelleneous
(15) EOP        - End of Procedure mark.  This opcode is never executed
                  but is output to mark the end of a procedure's code.
```

(End Table 1)

Language	Compiled (bytes)	Compile/load (seconds)	Execute (seconds)
Pascal MT+	308	102	19
Janus Ada	–	–	27
UCSD Pascal, Z-80	282	14	239
UCSD Pascal, TRS-80 Mod II	282	60	274
Augusta, Z-80 Osborne I	162	30	303
JRT Pascal V2.0	224	34.5	383
Supersoft Ada	–	–	422
Pascal/M	301	50	450
JRT Pascal	232	65	470
CBASIC2	–	26	484
Tiny-C Compiler	–	96	930

Table 2.

Comparison of Augusta with other languages. Chart indicates compiled program size and compilation/load and execute times for the Eratosthenes Sieve prime number generator. See references 1 and 2 (page 26) for details.

Listing 1.

Augusta Source Code for Prime Number Generation

```
 2    0      0      0 -- Eratosthenes Sieve Prime Number Program
 3    0      0      0 -- in Augusta
 4    0      0      0
 5    0      0      0 PROCEDURE PRIME IS
 6    1      0    -14   PRIME,K,COUNT : INTEGER;
 7    1      0      6   FLAGS : ARRAY(8190) OF BOOLEAN;
 8    1      0  16388
 9    1      0  16388 BEGIN
10    1      0  16388   PUTLINE("10 ITERATIONS");
11    1     17  16388   FOR  ITER IN 1..10
12    1     26  16388   LOOP
13    1     30  16390     COUNT := 0;
14    1     34  16390     FLAGS(0) := TRUE;  -- Initialize Array
15    1     40  16390     MOVELEFT ( @FLAGS(0), @FLAGS(1), 16378);
16    1     53  16390     FOR  I IN 0..8190
17    1     64  16390     LOOP
18    1     68  16392       IF  FLAGS(I)  THEN
19    1     78  16392         PRIME := I + I + 3;
20    1     90  16392         K := I + PRIME;
21    1     99  16392         WHILE K <= 8190
22    1    104  16392         LOOP
23    1    108  16392                 FLAGS(K) := FALSE;
24    1    115  16392                 K := K + PRIME;
25    1    123  16392         END LOOP;
26    1    123  16392         COUNT := COUNT + 1;
27    1    133  16392 --      PUTSTR("PRIME #=");
28    1    133  16392 --      PUTINT(PRIME);
29    1    133  16392 --      NEWLINE;
30    1    133  16392       END IF;
31    1    133  16392     END LOOP;
32    1    133  16392   END LOOP;
33    1    139  16390   PUTINT(COUNT); PUTLINE(" PRIMES");
34    1    160  16388 END;
```

Listing 2.

Disassembly of the Object Code Into P-Codes

```
Dissassembly of Augusta Program:  SIEVE.CPL
Number of Procedures =  1
-----------------------------------

Procedure #  1
Local Size =  16392
Parameter Size =  0
Lex Level =  1


 Offset      Opcode Value     Parameters
(     0) :     13  LCA     13 "10 ITERATIONS"
(    15) :     42  CSP     2
(    17) :      3  LLA     16388
(    20) :     65  SLDC1
(    21) :     11  STO
(    22) :      2  LDL     16388
(    25) :     74  SLDC10
(    26) :     27  LEQI
(    27) :     38  FJP     115  =>  145
(    30) :     58  SLA0    4
(    32) :     64  SLDC0
(    33) :     11  STO
(    34) :     58  SLA0    6
(    36) :     64  SLDC0
(    37) :     24  IND
(    38) :     63  SLDCN1
(    39) :     11  STO
(    40) :     58  SLA0    6
(    42) :     64  SLDC0
(    43) :     24  IND
(    44) :     58  SLA0    6
(    46) :     65  SLDC1
(    47) :     24  IND
(    48) :      1  LDCI    16378
(    51) :     42  CSP     12
(    53) :      3  LLA     16390
(    56) :     64  SLDC0
(    57) :     11  STO
(    58) :      2  LDL     16390
(    61) :      1  LDCI    8190
(    64) :     27  LEQI
(    65) :     38  FJP     71  =>  139
(    68) :     58  SLA0    6
(    70) :      5  LDO     16390
(    73) :     24  IND
(    74) :     12  SINDO
(    75) :     38  FJP     55  =>  133
(    78) :     58  SLA0    0
(    80) :      5  LDO     16390
(    83) :      5  LDO     16390
(    86) :     19  ADI
(    87) :     67  SLDC3
(    88) :     19  ADI
(    89) :     11  STO
(    90) :     58  SLA0    2
(    92) :      5  LDO     16390
(    95) :     57  SLDO    0
(    97) :     19  ADI
(    98) :     11  STO
(    99) :     57  SLDO    2
(   101) :      1  LDCI    8190
(   104) :     27  LEQI
(   105) :     38  FJP     18  =>  126
(   108) :     58  SLA0    6
(   110) :     57  SLDO    2
(   112) :     24  IND
(   113) :     64  SLDC0
(   114) :     11  STO
(   115) :     58  SLA0    2
(   117) :     57  SLDO    2
(   119) :     57  SLDO    0
(   121) :     19  ADI
(   122) :     11  STO
(   123) :     37  UJP     -27  =>  99
(   126) :     58  SLA0    4
(   128) :     57  SLDO    4
(   130) :     65  SLDC1
(   131) :     19  ADI
(   132) :     11  STO
(   133) :     80  INCL    16390
(   136) :     37  UJP     -81  =>  58
(   139) :     80  INCL    16388
(   142) :     37  UJP     -123  =>  22
(   145) :     57  SLDO    4
(   147) :     42  CSP     4
(   149) :     13  LCA     7 " PRIMES"
(   158) :     42  CSP     2
(   160) :     43  RET
```

End Listing

A Small-C Operating System

by Brian McKeon

Brian McKeon, C.N.P.U., Ward 4, Prince Henry Hospital, Little Bay, NSW, Sydney, Australia.

Copyright © 1983 by Brian McKeon. For non-commercial use only. Any commercial use without the author's written permission is prohibited.

Want to experiment with operating systems? Well, here is the source code of a system written in the programming language "C" and developed using Ron Cain's Small-C compiler (*DDJ* No. 45).

As I was lucky enough to have had access to a UNIX system, I was able to bring up the Small-C compiler with a minimum of fuss. After a couple of weeks of spare time, I was able to return to the microprocessor system and run the compiler. It was then that the contrast between the large UNIX system and a micro system became obvious. In particular, I missed the ability to redirect input/output via the operating system. With UNIX, any program output destined for the console can be redirected to go to a file. Console input to a program can also be arranged to come from a file. Both functions can be easily specified on the command line (see "echo" later).

Many of the operating systems for micros were developed in the days when 32Kbytes of memory was a large amount for a microprocessor system. Now it is by no means unusual for even personal machines to have 64Kbytes. A portion of this extra capacity could be devoted to making the micro operating system more flexible. Initially, I was simply trying to add routines to the micro to implement I/O redirection.

It was an ideal opportunity to try out the Small-C compiler. However, after a very short time I found that I seemed to have a great many of the routines of a full operating system. As an exercise, I then decided to have a go and write a full system and this article is the result.

I have incorporated the UNIX process of having directories as files themselves. While this may seem unusual, it does lead to advantages in that many of the system routines, available for file access, can be used to process directory information. User programs can just as easily access directories as files.

The system loads and executes user programs at 100 hex to make possible the execution of CP/M-compatible programs if the source code is not available. These programs would require some form of interface to translate system calls from CP/M convention to the form expected by the described system. As an alternative, the system calls described could be altered to give a CP/M-compatible interface. While it is possible to add such an interface, this may not be worth the effort. The book, *Software Tools*,[1] gives the source code of many useful programs, even a sophisticated editor. These are written in Ratfor but it is not a difficult task to translate to "C" and this would allow the use of more sophisticated system calls and make possible the transfer of programs to other systems.

In all of my work so far, I have used the Small-C compiler as published (*DDJ* Nos. 45, 48) with modifications enabling backlash escape sequences in literal characters (*DDJ* No. 56) and the removal of the bugs in the arithmetic rotate routines (*DDJ* No. 52). One of the more advanced versions now being sold would make the translation and compilation of any published "C" programs easier. Remember that source code will be required if you want to bring up the compiler on a new system.

Characteristics of this operating system are: input/output redirection on the command line; hierarchical directories (directories of directories and files); directories able to be processed as files; a copy in memory of the pointers used in file maintenance (giving fast file access); a disk group of 512 bytes (could be altered); or pointer range of 32767 disk groups; parsed command line arguments passed to user programs (see UNIX argc, argv[ ]); and dynamic memory allocation of 512-byte blocks.

A disadvantage of the system as it stands is that it has been developed for a one-disk-drive machine. Simultaneous access of multiple drives would require either swapping of the directory pointer blocks to/from the drives or maintenance of all pointer arrays in memory. While using more memory, the second alternative is the more time efficient. Allocating the top 16K of memory to the system program and the pointer arrays would allow the addressing of greater than 1Mbyte of disk space. The upper limit of disk space is the pointer range of 32767 disk groups or 16 megabytes with 512 byte group size. Large system size would make memory storage of arrays wasteful and an alternative file structure is described towards the end of this article that would be a good basis for larger capacity systems.

File Structure

There are three pointer arrays being swapped between disk and memory: glinks[ ], loptr[ ], and hiptr[ ] (see Listing on page 41). The largest array is glinks[ ]. For any group "i" on disk, the next group in the linked list from group "i" would be glinks[i]. If glinks[i] is zero, it indicates the end of a list of groups, i.e., end of a file (Figure 1, page 37). There is a linked list for each file, a list of free groups left on disk, and even a short linked list that the system uses to load and save an image of the arrays glinks-loptr-hiptr on disk.

Files are tied into a directory structure by the arrays loptr[ ] and hiptr[ ]. A directory is simply a linked list of files and this is achieved using the array loptr[ ] with loptr[k] used to index the next file in loptr[ ]. See Figure 2, page 37, for a pictorial representation of this and the following arrangements. If loptr[k] is zero, then the last file in the current directory has been reached. If the value of hiptr[ ] for a given directory element is positive, it is taken to be a pointer to the first group of a file, with the array glinks[ ] linking further groups of the file. If negative, this element of the directory is taken to be a further directory and the negative value is complemented and used to index back into the directory array loptr[ ].

The first entry in a directory is always a file of the *names* of files in that directory, one name per line. As an end-of-line character is used as a name delimiter, there is virtually no limit to name length. The *n*th line refers to the *n*th entry along the loptr[ ] list for that directory. If there are 32 lines in the file of filenames for a directory, then it must be possible to index through the loptr[ ] array 32 times and find a zero in that element of loptr[ ], i.e. the expected end of that directory list of files.

The first file name in a directory file is always a file called "." and is a reference to the file of filenames for that directory.

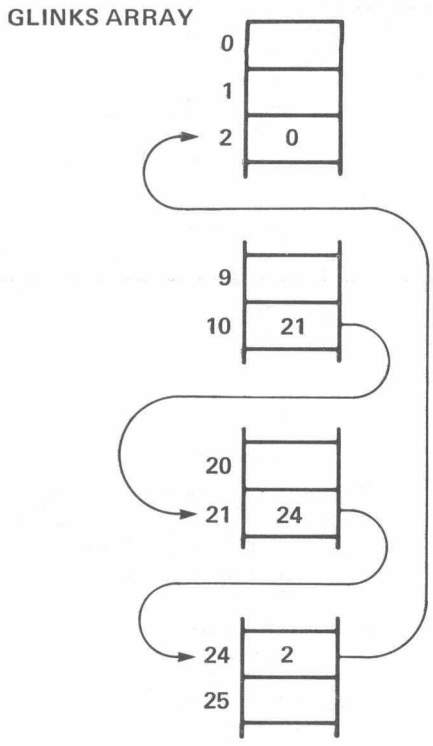


Figure 1.

Usage of array glinks[ ] to link groups of a file with first group number 10. The file consists of groups 10, 21, 24, and 2, in that order.

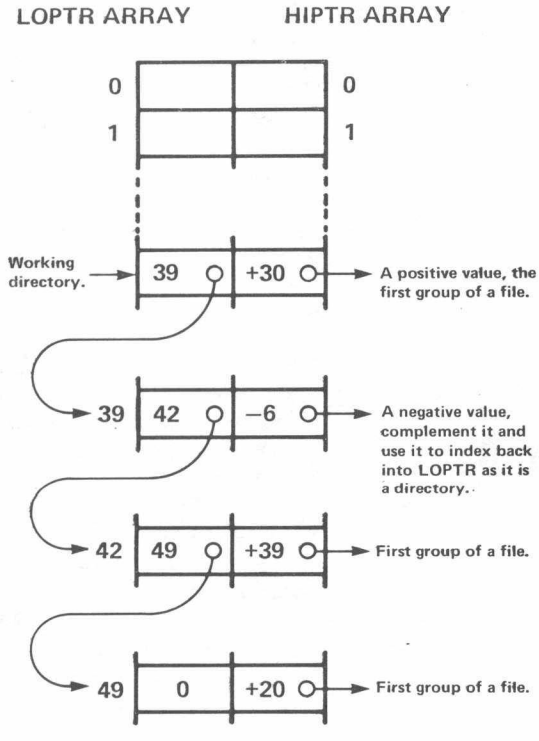


Figure 2.

An example of a directory consisting of four entries, the second of which is another directory, the others files.

On power-up, the system will always get loptr[2] (one of a few reserved locations) and assume that this pointer points to the user's directory. When the system opens a file, given the name, it scans the directory (see "scndir(name)," system call) and when the line number of the entry is known, it indexes into the loptr[ ] array this number of times. If the hiptr of this entry is negative, the entry must be another directory; if positive, it is the first disk group of a file entry.

As already mentioned, redirection of console input and output has been incorporated in this sytem. When the character ">" is encountered on the command line, the next characters are assumed to be a filename, a file of that name is created, and any program output that would have been put to the console will go to that file. Similarly, the character "<" indicates that any requests by the executed program for console input will be taken from the file whose name follows the "<" character.

System Overview

The flexibility of the overall system becomes obvious when the simple program "echo" is considered:

```
main()
    {while(putchar(getchar()));}
```

Execution of this program echoes typed characters. They will be doubly echoed, as I have written getchar to echo any non-control characters. If, on the command line, we say "echo >newfile," any characters typed at the console will be redirected into the file "newfile." If a file by that name already exists, then it will be replaced. To terminate the process, type control d. This gives the end of file code. Then, on the command line "echo<newfile"; the file will then be listed to the console, as the input to echo has been redirected from console to come from "newfile." "echo <newfile >newerfile" gives a copy of newfile into newerfile. Now "echo <.", the contents of the file ".", which is the file of filenames for the current working directory, will be shown.

When the user changes his working directory using the system call "chdir," the value of loptr[ ] for the new directory is stored in loptr[3]. The default directory for the user is in loptr[2]. If "chdir" is called with a zero argument, the user is returned to his default directory from wherever he is. If a program to be executed cannot be found in the working directory, the system directory will be searched by default. The system directory is pointed to by loptr[1].

A list of the system calls is in Table 1. The listing of "library for Small-C system programs" gives details of how getchar and putchar are formed. A function that illustrates some system calls and is also required by the Small-C compiler on the new system is that of "fopen" (note that I have altered the second argument to be an "int").

```
fopen(name,func)
    char name[];
    int func;
    {
    int buffa;
    if((buffa=alloc())==0)return 0; /* get a buffer */
    if(func==rdfn)return (openf(name,buffa,rdfn));
    if(func==wrfn)return (creatf(name,buffa));
    dalloc(buffa); /* error-give buffer back */
    return 0;
    }
```

System Transfer

It is not enough to have a copy of the system code on the disk, as the initial directory also needs to be there and, as there are no intrinsic functions in the system, there should be at

least a memory to disk save program on the disk as well. My memory save program takes a numeric argument and saves that number of groups beginning at a predetermined location. One way to get going would be to put sectors onto the disk manually but there is another easier way. I started debugging the system by assembling it down low in memory and having the disk I/O routines simply doing block moves to and from higher memory. Simple programs and text could be loaded into memory at the appropriate location and loaded by the Small-C system as if coming from disk.

At this stage, the directory loader routine generated a dummy directory with three files, each of four groups (2K), a free group list and a free pair (loptr-hiptr) list. The free group list is simply a long file pointed to by hiptr[2]. The free pair list is a linked list of any spare loptr[ ] cells and is used when a new file or directory is created. This linked list of cells is pointed to by hiptr[1]. When the bugs were out of the system, I altered the disk write routine to actually write information to the disk. I then had a suitably formatted disk for the new system. After copying this disk (twice), I recompiled the system to reside in high memory, altered the disk read to come from disk instead of memory, and the system was up.

The system code takes up about 9.5K and the pointer buffers about 2.5K for my 250K drive. One system call I will be changing in the future is "scndir." Filenames will be allowed to contain "?" (match with any character) and "*" (match with any number of characters). A second parameter will also be passed and this will tell scndir what line/entry in the directory to begin the search with. Another area that I will be modifying will be the system of group linking for disk files. The group links will be part of the files on disk eliminating the large array "glinks" (Figure 3).

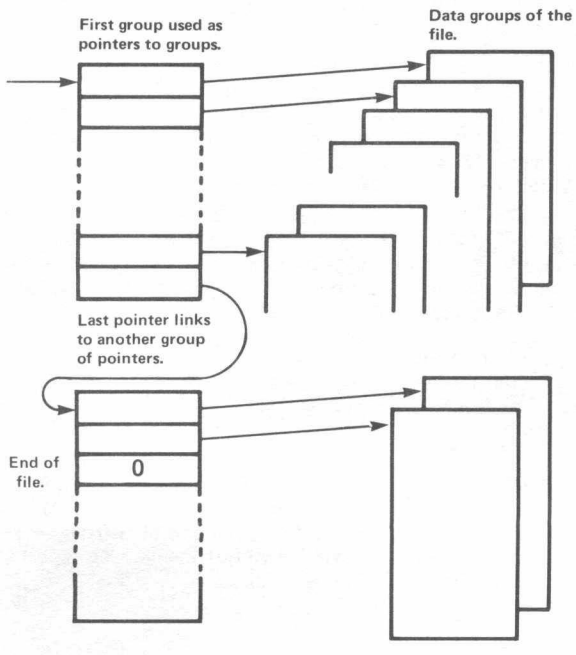


Figure 3.
A proposed file structure that would remove the need for an in-memory array of group links (glinks).

System calls	Descriptions
puterr(str)	Put a string to the console (cannot be redirected, used for error messages).
getc(unit)	Get a character from "unit," unit=0 standard input, unit=1 the communication port, returns character or 0 on end of file.
putc(c,unit)	Put a character to "unit" (see getc), returns "c".
readf(unit)	Read the next sequential group on opened file, returns 0 on end of file.
writef(unit)	Write the next sequential group.
openf(name,buffer,fn)	Open "name" for read(fn=1) write(fn=2) or append (fn=3) with "buffer" as the buffer address, returns a unit number or 0 if not found.
create(name,buffer)	As for openf, write assumed, any old file by that name is lost.
closf(unit)	Close the given unit number, flushes write buffer if required.
readr(unit,N)	Read *n*th group from opened unit number, returns 0 if out of file group range.
writer(unit,N)	Write *n*th group to opened unit number, returns 0 if out of range.
scndir(name)	Scan the working directory for "name," returns number of entry in directory, 1 to *n* or 0 if not there.
chdir(name)	Look for "name" in the current directory and if it is a directory, change operations to it, if name==0 changes back to root directory of user.
mkdir(name)	Create a new directory "name," returns a 0 if name in use.
rmfil(name,i)	Remove directory entry "name," if not found, returns 0. Won't allow rmfil("."). If "i" is 0, won't allow directory entry removal. If "i" is 1, a recursive remove of the entry is undertaken, directory or file.
alloc()	Return the address of the next free top 512 byte block of memory. Addr will lie on a 512 byte boundary.
dalloc(addr)	Give back a 512 byte block of memory for future use.
dirfn(i)	For directory information: i=0 returns ptr to *glinks[ ], i=1 returns* ptr to loptr[ ], i=2, returns ptr to hiptr.
dirio(rw)	Refresh or save the directory pointer arrays: rw=1 (write) gives a save to disk, rw=2 (read) gives a refresh from disk.

Table 1.
System Calls.

Conclusion

While I have a compiler on the system (courtesy of Ron Cain), I do not yet have an assembler for it. If any reader has written an assembler (in Small-C?) for 8080 mnemonics, I would like to hear from them. Good luck!

Listing begins on next page.

DDJ

Editor's Note: For those who have not yet implemented the Small-C compiler published in DDJ, *some back issues are still available. The greatly improved Small-C v.2, by J.E. Hendrix, is available in issues #74 and #75. Ron Cain's version one of the compiler, along with bug fixes and patches, can most easily be found in our Bound Volumes.*

Reference

[1] Kernighan, B.W. and Plauger. *Software Tools.* Addison-Wesley, Reading, Massachusetts, 1976.

Small-C Operating System (Text begins on page 36)

Listing One

```
/** operating system written in Small-C     **/
/** Brian McKeon 5/5/82           **/
#define eol 10          /* end of line char */
#define bufsiz     512  /* disk buffer size */
/** some memory control           **/
#define usrprg 256 /* start and entry point of user program */
int memmap[8];          /* bitmap of 512 byte blocks of memory */
/** and directory structure       **/
#define dirone 22  /* ptr to 1st grp of glinks-loptr-hiptr */
#define lnksiz 500 /* one link per 'bufsiz' bytes */
int glinks[lnksiz];
#define ptrsiz 50  /* seems to be enough */
int loptr[ptrsiz];
int hiptr[ptrsiz];
/** First few pointer pairs have special functions **/
/** zero ptrs are set to to 0 as traps              **/
#define rsvptrs 4  /* reserved system pointer pairs */
#define sysdir loptr[1] /* ptr to system dir struct */
#define userdir loptr[2] /* ptr to root dir of user */
#define workdir loptr[3] /* ptr to -current- user dir structure */
#define freptr hiptr[1] /* ptr to free dotted pair list */
#define grpfree hiptr[2] /* ptr to first free disk group */
/** i/o structures           **/
int stdout;        /* and for standard output */
#define maxunits 10     /* max no. simul open files */
#define minunit 2  /* 0, 1 reserved for console and communication port */
int grw[maxunits]; /* 0 if free, 'rdfn' for read and 'wrfn', write */
#define rdfn  1
#define wrfn  2
#define apfn  3    /* append, used for openf call only */
/** structures to keep track of file i/o **/
int firgrp[maxunits];   /* first group of a file group list */
int lnkgrp[maxunits];   /* link group for sequential i/o */
int gbuffs[maxunits];   /* addr of the buffers assoc with a unit */
int gptrs[maxunits];    /* pointers into these buffers */
char sysinp[bufsiz];    /* system input buffer */
char sysout[bufsiz];    /* and output buffer */
/** command line argument passing to user programs **/
int argc;          /* argument count */
int argv[10];      /* array of ptrs to argument strings */
/** line input buffer - argv[] strings **/
#define maxlin 128 /* max line input size */
char linbuf[maxlin];    /* line input buffer */
/** main entry point for system **/
/* call the system with the desired i/o parameters */
/* with arg0 the number of the system call  */
/*                          */
#asm
    org 0a000h
#endasm
             /* 0a000 gives blocks 1 to 79 spare */
#define maxblk 77  /* leave a couple of blocks for stack */
system(arg0,arg1,arg2,arg3)
    int arg0,arg1,arg2,arg3;
```

```
{
int munit,savedir;
if(arg0==0)return (puts(arg1));     /* put str to std err */
if(arg0==1)return (getc(arg1));     /* char fetch */
if(arg0==2)return (putc(arg1,arg2)); /* char put */
if(arg0==3)return (readf(arg1));    /* sequential read */
if(arg0==4)return (writef(arg1));   /* sequential write */
if(arg0==5)return (openf(arg1,arg2,arg3)); /* open a file */
if(arg0==6)return (creatf(arg1,arg2)); /* create a file */
if(arg0==7)return (closf(arg1));    /* close a unit */
if(arg0==8)return (readr(arg1,arg2)); /* read random grp */
if(arg0==9)return (writer(arg1,arg2)); /* write random */
if(arg0==10)return (scndir(arg1)); /* scan dir for file name */
if(arg0==11)return (chdir(arg1));   /* change directory */
if(arg0==12)return (mkdir(arg1));   /* make new dir */
if(arg0==13)return (rmfil(arg1,arg2));  /* remove file-directory */
if(arg0==14)return (alloc()); /* get addr of free blk */
if(arg0==15)return (dalloc(arg1)); /* de-alloc a mem blk */
if(arg0==16)return (dirfn(arg1));   /* give addr of dir in mem */
if(arg0==17)dirio(wrfn); /* go system, save the dir */
else {inits();dirio(rdfn);
     puts("illegal system call >17: directory restored\n");
     }
while(1)  /* command interpreter loop */
     {
     inits();       /* re-initialize stk, i/o, hardware */
     puts("$ ");          /* prompt to user */
     if((munit=getlin(linbuf))==0)continue; /* get command */
     if(parslin(linbuf,munit)==0)  /* parse into argc, argv */
          continue;       /* on parse error */
     if((munit=openf(argv[0],sysinp,rdfn))==0)
          {
          if(workdir==sysdir){puts("open error\n");continue;}
          else
               {
               savedir=workdir; /* save the working dir */
               workdir=sysdir; /* and look in the sys dir */
               munit=openf(argv[0],sysinp,rdfn);
               workdir=savedir;
               if(munit==0){puts("open error\n");continue;}
               }
          }
     if(loadf(munit)==0)continue; /* and load the file */
     closf(munit);       /* close the loader unit */
     usrprg(argc,argv);  /* call the loaded user program */
     closf(stdout);      /* flush the output if any */
     dirio(wrfn);        /* and save the dir */
     }
}

/* get the next line of input from the console */
/* all white space is replaced by nulls (0) for parsing */
#define backsp 8
getlin(buffa)
    char buffa[];
```

(Continued on next page)

```
    {
    int k;
    char c;
    k=0;
    while(k<maxlin)
        {
        c=inchar();
        if(c==eol)break;
        if(c==backsp)
            {
            if(k){--k;outchar(c);}
            continue;
            }
        if(c<=' ')c=0;
        buffa[k++]=c;
        }
    buffa[k]=0;
    return k;
    }
/* parse a line into argc, argv and also re-direct i/o */
parslin(str,nchar)
    char str[];
    int nchar;
    {
    int pt,endarg;
    char lastc;
    pt=argc=0;
    argv[0]=&str[0];
    while(1)
        {
        while(str[pt])++pt; /* move over argument */
        while(str[pt]==0)++pt;   /* and nulls */
        if(pt>nchar)break;  /* finished? */
        if(str[pt]=='>')    /* check for re-direction */
            {
            if((stdout=creatf(&str[pt+1],sysout))==0)
                {puts("i/o redirection error\n");return 0;}
            }
        else if(str[pt]=='<')
            {
            if((stdin=openf(&str[pt+1],sysinp,rdfn))==0)
                {puts("i/o redirection error\n");return 0;}
            }
        else argv[++argc]=&str[pt];
        }
    return ++argc;
    }

/* open a given file name for file i/o */
/* from the current directory          */
/* rw is 'rdfn' for input, 'wrfn' for output and 'apfn' for append */
openf(name,buffer,rw)
```

```
        char name[],buffer[];
        int rw;
        {
        int filno;
        if((filno=sendir(name))==0)return 0;  /* find file name */
        return (openit(filno,buffer,rw));
        }

/* lower level open operates from number entry of file in dir */
/* for example 1 would open the dir file itself              */
/* the append function should only be used with null terminated */
/* files as it searches for this null char.      */
openit(filno,buffer,rw)
    int filno;
    char buffer[];
    int rw;
    {
    int i,k,grp;
    i=workdir;
    while(--filno)i=loptr[i]; /* down the dir list */
    if(hiptr[i]<0)return 0;    /* attempting to open dir? */
    if((filno=findio())==0)return 0; /* any i/o units left? */
    firgrp[filno]=hiptr[i];  /* pointer to first grp of file */
    gbuffs[filno]=buffer; /* setup buffer location */
    if(rw==rdfn)        /* read ? */
         {
         grw[filno]=rdfn;
         lnkgrp[filno]=hiptr[i];/* read from this first group */
         gptrs[filno]=bufsiz;     /* initialize the pointer */
         }
    else if(rw==wrfn)        /* write ? */
         {
         frefil(hiptr[i]);   /* free up the old groups */
         hiptr[i]=0;         /* initially append null */
         grw[filno]=wrfn;    /* write */
         lnkgrp[filno]=-i;   /* by -, indicate a link to the dir! */
         gptrs[filno]=0;          /* initialize the pointer */
         }
    else if(rw==apfn)        /* append ? */
         {

         /* get to the last group */
         grp=hiptr[i];
         if(glinks[grp]==0)lnkgrp[filno]=-i;
         else
              {
              while(glinks[glinks[grp]]!=0)grp=glinks[grp];
              lnkgrp[filno]=grp; /* 2nd last grp */
              grp=glinks[grp];   /* last grp */
              }
         if(grupio(grp,buffer,rdfn)==0)return 0; /* read last grp */
         frefil(grp);      /* and give it up */
         grw[filno]=wrfn;
         k=0;
```

(Continued on next page)

```
            while(buffer[k]!=0)++k;  /* go to the old eof */
            gptrs[filno]=k;      /* pointer now ready for append */
            }
        return filno;
        }
    /* create a file */
    creatf(name,buffa)
        char name[],buffa[];
        {
        int unit,k,l,newfree;
        if((grpfree==0)|(freptr==0)|(name[0]==0))return 0;
        if(unit=openf(name,buffa,wrfn))return unit; /* in the dir? */
        if((unit=openit(1,buffa,apfn))==0)return 0; /* no, open the dir */
        k=workdir;          /* and append the file */
        while(loptr[k])k=loptr[k];    /* go down the dir list */
        newfree=loptr[freptr];   /* get a new free pointer */
        loptr[k]=freptr;     /* and add to the directory */
        loptr[freptr]=0;     /* append a null */
        hiptr[freptr]=0;     /* and no groups */
        freptr=newfree;          /* up-date free pointer */
        l=0;      /* errors after this point are fatal */
        while(name[l]) /* append the name */
            if(putc(name[l++],unit)==0)   /* to the dir file */
                createrr(k);
        if(putc(eol,unit)==0)
            createrr(k);
        closf(unit);   /* and close the directory */
        return (openf(name,buffa,wrfn)); /* openf should now work */
        }

    /* fatal error routine from create function */
    /* dont flush the dir buffer as there is an error somewhere */
    createrr(k)
        int k;
        {
        int oldfree;
        oldfree=freptr;
        freptr=loptr[k];
        loptr[k]=0;    /* re-terminate the dir */
        loptr[freptr]=oldfree;
        puts("fatal failure on file create\n");
        system(17,0,0,0);   /* and try to recover */
        }

    /* close a given unit number */
    closf(unit)
        int unit;
        {
        int k;
        k=1; /* default return value */
        if(unit<minunit)return 0;     /* trap illegal close */
        if((grw[unit]==wrfn)&(gptrs[unit]!=0))  /* write required? */
            k=writef(unit);
        grw[unit]=0;   /* now free up this unit */
        return k;
        }
```

```
/* look for a spare unit number else return 0 */
findio()
    {
    int k;
    k=minunit;
    while(k<maxunits)
         {if(grw[k]==0)return k;++k;}
    return 0;
    }

/* scan directory for a matching line to the given one */
/* and return a number 1 to N, else zero if not found */
scndir(name)
    char name[];
    {
    int linnum,munit,k,flg;
    char c;
    if(name[0]==0)return 0;
    if((munit=openit(1,sysinp,rdfn))==0)return 0; /* open dir for read */
    linnum=0;
    while(1)
         {
         flg=1;
         k=0;
         ++linnum;
         while(1)
              {
              if((c=getc(munit))==0){closf(munit);return 0;}
              if(c==eol)break;
              if(c!=name[k++])flg=0;
              }
         if((name[k]==0)&flg){closf(munit);return linnum;}
         }
    }

/* put an error message to the console */
puts(str)
    char str[];
    {
    int j;
    j=0;
    while((outchar(str[j++]))!=0);
    }

/* get a character from a given unit number */
getc(unit)
    int unit;
    {
    char c;
    if(unit==0)unit=stdin;   /* standard in? - re-direct */
    if(unit==0)return (inchar()); /* console port */
    if(unit==1)return (incom());  /* communication port */
    if(grw[unit]!=rdfn)return 0;  /* open for read? */
    if(gptrs[unit]==bufsiz)       /* get pointer - empty? */
         if(readf(unit)==0)return 0;
    c=*(gbuffs[unit]+gptrs[unit]);
    ++gptrs[unit];
    return c;
    }
```

(Continued on next page)

```
/* sequential read of file */
readf(unit)
    int unit;
    {
    int nxtgrp;
    if(grw[unit]!=rdfn)return 0;
    gptrs[unit]=0;
    nxtgrp=lnkgrp[unit];
    lnkgrp[unit]=glinks[nxtgrp];  /* update grp no. */
    return (grupio(nxtgrp,gbuffs[unit],rdfn)); /* refill buffer */
    }
/* random read of file group, range 0 to N */
readr(unit,grpno)
    int unit,grpno;
    {
    int rdgrp;
    if(grw[unit]!=rdfn)return 0;
    rdgrp=firgrp[unit];
    while(grpno--)rdgrp=glinks[rdgrp]; /* down the file grp list */
    return (grupio(rdgrp,gbuffs[unit],rdfn));
    }
/* put a character to a given i/o unit */
putc(c,unit)
    char c;
    int unit;
    {
    if(unit==0)unit=stdout;       /* standard out? - redirect */
    if(unit==0)return (outchar(c)); /* console port */
    if(unit==1)return (outcom(c)); /* communication port */
    if(grw[unit]!=wrfn)return 0;  /* write unit? */
    if(gptrs[unit]==bufsiz)       /* full buffer? */
        if(writef(unit)==0)return 0;
    *(gbuffs[unit]+gptrs[unit])=c;
    ++gptrs[unit];
    return c;
    }
/* write sequential to given unit */
writef(unit)
    int unit;
    {
    int newgrp;
    if(grw[unit]!=wrfn)return 0;
    gptrs[unit]=0;
    newgrp=fregrp();    /* get a new group */
    if(lnkgrp[unit]>0)glinks[lnkgrp[unit]]=newgrp; /* link */
    else hiptr[-lnkgrp[unit]]=newgrp; /* special case: link to dir */
    lnkgrp[unit]=newgrp;     /* and ready for next call */
    return (grupio(newgrp,gbuffs[unit],wrfn));
    }
/* random write of file group, range 0 to N */
writer(unit,grpno)
    int unit,grpno;
    {
    int wrgrp;
    if(grw[unit]!=wrfn)return 0;
```

```
    wrgrp=firgrp[unit];
    while(grpno--)wrgrp=glinks[wrgrp]; /* down the file grp list */
    return (grupio(wrgrp,gbuffs[unit],wrfn));
    }
/* free up a list of groups */
frefil(grp)
    int grp;
    {
    int fgrp; /* a pointer to the first free group */
    if(grp==0)return 0; /* dont lose the free list */
    fgrp=grpfree;  /* a pointer to the first free group */
    grpfree=grp;   /* point free list at the file */
    while(glinks[grp])grp=glinks[grp]; /* go to end of file */
    glinks[grp]=fgrp;   /* and link on the old free list */
    }
/* get a free group */
/* null terminate */
fregrp()
    {
    int grp;
    grp=grpfree;   /* find the next free group */
    grpfree=glinks[grp];    /* update the free pointer */
    glinks[grp]=0; /* and null terminate the returned group link */
    return grp;
    }

/* load a program, checking for oversize and for jump at start */
loadf(unit)
    int unit;
    {
    int grp,addr,blk;
    char ujump;
    grp=lnkgrp[unit];   /* get the first group */
    addr=usrprg:        /* load into the user area */
    blk=addr>>9;        /* and allocate mem as you go */
    while(grp)
        {
        if(blk==maxblk){puts("program too large\n");return 0;}
        if(grupio(grp,addr,rdfn)==0){puts("load error\n");return 0;}
        addr=addr+bufsiz;
        fixblk(++blk); /* allocate the mem */
        grp=glinks[grp];
        }
    ujump=*usrprg; /* should be the jump byte if a program */
    if(ujump!=-61){puts("not a program\n");return 0;}
    return 1;
    }
```

(Continued on next page)

```
/* save/restore the directory structure to/from disk */
dirio(rdwr)
    int rdwr;
    {
    int grp,addr;
    grp=dirone;
    addr=glinks;
    while(grp)
         {
         if(grupio(grp,addr,rdwr)==0)return 0;
         grp=glinks[grp];
         addr=addr+bufsiz;
         }
    }
/* change work directory */
chdir(name)
    char name[];
    {
    int pt,n;
    if(name==0){workdir=userdir;return;}
    if((n=scndir(name))==0)return 0;
    pt=workdir;
    while(--n)pt=loptr[pt];  /* down the dir list */
    if(hiptr[pt]>=0)return 0; /* not a directory */
    workdir=-hiptr[pt]; /* it is a dir, change */
    }
/* make a new directory */
mkdir(name)
    char name[];
    {
    int pt,n,newfree;
    if(n=openf(name,sysinp,rdfn))
         {closf(n);return 0;}     /* name is in use ! */
    if((n=creatf(name,sysinp))==0)return 0; /* cannot create */
    closf(n); /* got the name entry, close it */
    pt=workdir;
    while(loptr[pt])pt=loptr[pt]; /* name must be last on dir-list */
    newfree=loptr[freptr];   /* get the new free pointer */
    hiptr[pt]=-freptr;  /* link on a cell (- to indic dir) */
    pt=freptr;
    freptr=newfree;          /* update the freptr */
    loptr[pt]=0;   /* empty dir */
    hiptr[pt]=fregrp(); /* and the directory file grp itself */
    return (grupio(hiptr[pt],".\n",wrfn)); /* dir contains '.' only */
    }              /* i.e. itself */
/* remove a directory-files structure */
rmfil(name,flg)
    char name[];
    int flg;
    {
    int n,i,preptr,pt,postptr;
    if((n=scndir(name))==0)return 0; /* in the dir? */
    i=n;
    --i;
    if(i==0)return 0;   /* dont allow rmfil(".",x) */
```

```
    preptr=workdir;          /* and go to the link before the entry */
    while(--i)preptr=loptr[preptr];
    pt=loptr[preptr];   /* the entry */
    postptr=loptr[pt];  /* and the one after */
    if((hiptr[pt]<0)&(flg!=1))return 0; /* remove dir? */
    if(rmname(n)==0)return 0; /* now remove the n th. name */
    loptr[preptr]=postptr;        /* link around */
    loptr[pt]=0;        /* and terminate to restrain 'remove' */
    remove(pt);
    return 1;
    }
/* remove the n th. name from the work directory */
rmname(n)
    int n;
    {
    int obuf,ibuf,ounit,iunit,l,wrtflg;
    char c;
    wrtflg=obuf=iunit=ounit=0;
    while(1)  /* but only used for one-pass flow control */
         {
         if((ibuf=alloc())==0)return 0;
         if((obuf=alloc())==0)break;
         if((iunit=openit(1,ibuf,rdfn))==0)break;
         if((ounit=openit(1,obuf,wrfn))==0)break;
         l=1;
         wrtflg=1;      /* writing flag */
         while(c=getc(iunit))
              {
              if(n==l)wrtflg=0;   /* suppress this entry? */
              if(wrtflg)putc(c,ounit);
              if(c==eol){++l;wrtflg=1;}
              }
         putc(0,ounit);
         wrtflg=1;break;
         }
    closf(iunit);closf(ounit);dalloc(ibuf);dalloc(obuf);
    return wrtflg; /* will be 0 on an error */
    }
/* recursive remove of dir-file structure */
remove(pt)
    int pt;
    {
    int newfree;
    while(pt) /* down the dir list */
         {
         if(hiptr[pt]<0)remove((-hiptr[pt]));    /* if a dir */
         else frefil(hiptr[pt]);       /* or if a file */
         newfree=pt;    /* now add this loptr.hiptr to free list */
         pt=loptr[pt];  /* pt onto next in list */
         loptr[newfree]=freptr; /* link on the old free list */
         freptr=newfree;     /* and update the free pointer */
         }
    }
/* return the addr of the next free top blk of mem */
alloc()
    {
    int blk;
    blk=maxblk:
```

(Continued on next page)

```
    while(blk)
         if(blkfree(--blk))break;
    fixblk(blk);   /* fix this block */
    return (blk<<9);    /* and return the addr */
    }
/* fix a block of mem for use, given the block no */
fixblk(blk)
    int blk;
    {
    int bits,words;
    words=blk>>4;
    bits=blk-(words<<4);
    bits=-(1+(1<<bits));     /* complement for mask */
    memmap[words]=bits&memmap[words];
    }
/* is the given block number free ?: returns 1 if true, else 0 */
blkfree(blk)
    int blk;
    {
    int bits,words;
    words=blk>>4;
    bits=blk-(words<<4);
    return ((1<<bits)&(memmap[words]));
    }
/* free up the given block corresponding to the given addr */
dalloc(addr)
    int addr;
    {
    int bits,words,blk;
    addr=32767&(addr>>1);    /* take care to mask */
    blk=addr>>8;         /* and then calc the blk no. */
    if(blk==0)return;    /* trap */
    words=blk>>4;
    bits=blk-(words<<4);
    memmap[words]=memmap[words]|(1<<bits);
    }
/* return the location of a directory structure */
dirfn(arg)
    int arg;
    {
    if(arg==0)return glinks;
    if(arg==1)return loptr;
    if(arg==2)return hiptr;
    }
inchar()
    {
    char c;
    c=conin();     /*the machine code char input */
    if(c>=' ')outchar(c);    /*if not a control char */
    if(c==13)return outchar(eol); /* carr-ret to eol */
```

```
    if(c==4)return 0;    /* control-d ret null */
    return c;
    }
outchar(c)
    char c;
    {
    if(c==eol)conout(13);    /* insert a carriage return if eol */
    return (conout(c));
    }
#include shlibr.c
/* this library is simply the routines ccgchar-cmpbcde from DDJ 48 */
```

Listing Two

```
/* initialize stack, i/o and hardware etc. */
inits()
    {
    int i;
    #asm
    mvi  a,0c3h    ;fix up the system call location at zero
    sta  0
    lxi  h,system
    shld 1
    lxi  h,system-4
    pop  b    ;the local variable 'i'
    pop  b    ;and the return addr
    sphl
    push b    ;the return addr
    push b    ;and 'i'
    #endasm
    stdin=stdout=0;     /* std in, out via console */
    i=8;      /* now book up all memory */
    while(i)memmap[--i]=0;   /* and free up user area */
    i=maxblk;
    while(--i)dalloc(i<<9);  /* leaves zero block not free */
    i=0;
    while(++i<maxunits)grw[i]=0;
    loptr[0]=hiptr[0]=glinks[0]=0;     /* trap errors */
    }
/* get/put a group to disk        */
/* to/from a given dma address        */
/* returns 1 if o.k., 0 if error  */
#define sectrk 26       /* 26 by 128 byte sectors per track */
                /* conversions in the next section */
                /* assume a 512 byte bufsiz !!!       */
grupio(group,addr,io)
    int group,addr,io;
    {
    int track,sector,memad;
    if(group==0)return 0;
    track=(group<<1)/13;
    sector=(group<<2)-track*26; /* 0 to 25 range not 1 to 26! */
/***** insert routines to read/write 4 sequential sectors ******/
    }
```

(Continued on next page)

Small-C Operating System (Listing continued, text begins on page 36)

```
The following routines are some programs to get the system going.
The first is 'save'.

/* Save a specified number of blocks of memory into a specified file */
/* The blocks of memory are assumed to start at 2560 (0A00H)         */
#define bufsiz 512
char buff[bufsiz];
main(argc,argv)
    int argc,argv[];
    {
    int unit,ngrp,grp,addr;
    if(argc!=3){puterr("\nWrong no. of args");return;}
    if((unit=creatf(argv[1],buff))==0)
         {puterr("\nCannot open file");return;}
    if((ngrp=todec(argv[2]))==0){puterr("\nHow many blocks?");return;}
    addr=2560;      /* first location saved */
    while(ngrp--)
         {
         move(addr,buff);
         if(writef(unit)==0){puterr("\nWrite error");return;}
         addr=addr+bufsiz;
         }
    closf(unit);
    puterr("\nSuccessful save");
    }
/* move a buffer in memory */
move(from,to)
    char from[],to[];
    {
    int k;
    k=0;
    while(k<bufsiz)
       to[k]=from[k++];
    }
/* convert a string to an integer: return 0 on error */
/* routine straight from small-c compiler   */
todec(str)
    char str[];
    {
    int k,n,c;
    n=k=0;
    while(c=str[k++])
         {
         c=c-'0';
         if((c<0)|(c>9))return 0;
         n=10*n+c;
         }
    return n;
    }
#include syslib.c
End of program.

This program 'rm' enables the removal of files.
It is set up so as to be not able to remove directories.
#include syslib.c
```

```
main(argc,argv)
    int argc,argv[];
    {while(--argc)rmfil(argv[argc],0);}
End of program.

The next program is 'stats'.
/* Gives statistics of number of free dotted pairs and groups left */
/* If any arguments are given it will give the size of these files */
#include syslib.c
main(argc,argv)
    int argc,argv[];
    {
    int pt,n,i,glinks[],loptr[],hiptr[];
    glinks=dirfn(0);
    loptr=dirfn(1);
    hiptr=dirfn(2);
    i=1;
    while(i<argc)       /* for any given file names */
         {
         if((n=sendir(argv[i]))==0)continue;
         pt=loptr[3];   /* the working dir ptr */
         while(--n)pt=loptr[pt];  /* go to the files dotted pair */
         pt=hiptr[pt];  /* a pointer to the file */
         if(pt>0)
              {
              n=size(pt,glinks);
              if(n<0)puts(" corrupted file structure in ");
              else {outdec(n);puts(" groups in");}
              puts(argv[i++]);puts("\n");
              }
         }
    pt=hiptr[1];
    n=size(pt,loptr);   /* find the no. of free nodes */
    if(n<0)puts(" free node list is corrupted\n");
    else outdec(n);puts(" free nodes left\n");
    pt=hiptr[2];
    n=size(pt,glinks);  /* find the no of free grps left */
    if(n<0)puts("\n free group list is corrupted\n");
    else outdec(n);puts(" free groups left\n");
    }
size(pt,array)
    int pt,array[];
    {
    int n;
    n=0;
    while(pt)
         {
         pt=array[pt];
         if(++n>32000){n=-1;break;}
         }
```

(Continued on next page)

Small-C Operating System

(Listing continued, text begins on page 36)

```
    return n;
    }
puts(str)
    char str[];
    {
    int k;
    k=0;
    while(putchar(str[k++]));
    }
outdec(num)
    int num;
    {
    int k,zs;
    char c;
    zs=0;
    k=10000;
    if(num<0){num=(-num);putchar('-');}
    while(k>=1)
        {
        c=num/k+'0';
        if((c!='0')|(k==1)|(zs))
            {zs=1;putchar(c);}
        num=num%k;
        k=k/10;
        }
    }

/* library for small-c system programs */
#asm
sys equ  0
#endasm

/* some definitions */
#define rdfn 1    /* file read function */
#define wrfn 2    /* write function */
#define apfn 3    /* and append */
#define bufsiz 512
#define eol 10
puterr(str)
    char str[];
    {sys(0,str,0,0);}
getchar()
    {sys(1,0,0,0);}
putchar(c)
    char c;
    {sys(2,c,0,0);}
getc(unit)
    int unit;
    {sys(1,unit,0,0);}
putc(c,unit)
    char c;
    int unit;
    {sys(2,c,unit,0);}
readf(unit)
    int unit;
    {sys(3,unit,0,0);}
writef(unit)
    int unit;
    {sys(4,unit,0,0);}
openf(name,buff,fn)
    char name[],buff[];
    int fn;
    {sys(5,name,buff,fn);}
creatf(name,buff)
    char name[],buff[];
    {sys(6,name,buff,0);}
closf(unit)
    int unit;
    {sys(7,unit,0,0);}
readr(unit,n)
    int unit,n;
    {sys(8,unit,n,0);}
writer(unit,n)
    int unit,n;
    {sys(9,unit,n,0);}
scndir(name)
    char name[];
    {sys(10,name,0,0);}
chdir(name)
    char name[];
    {sys(11,name,0,0);}
mkdir(name)
    char name[];
    {sys(12,name,0,0);}
rmfil(name,flg)
    char name[];
    int flg;
    {sys(13,name,flg,0);}
alloc()
    {sys(14,0,0,0);}
dalloc(addr)
    int addr;
    {sys(15,addr,0,0);}
dirfn(arg)
    int arg;
    {sys(16,arg,0,0);}
gosys()
    {sys(17,0,0,0);}
#asm
;fetch a single byte from the address in
hl into hl  The routines are as in DDJ 48
but only the routines'ccgchar' to 'cmpbcde'
#endasm
```

End Listing

6809 Threaded Code

Parametrization and Transfer of Control

by H. T. Gordon

H. T. Gordon, College of Natural Resources, University of California, Berkeley, CA 94720.

High-level languages of the Forth type – now proliferating like the rabbits that inspired creation of Fibonacci numbers – are usually not compiled to machine-language "object" code. Their more-or-less humanly intelligible source code gets translated by a text-interpreter program into an intermediate code ("intcode"), a compact sequence of memory addresses (that point to a block of machine code) and parameters. This gets executed by an inner interpreter program that transfers control successively to the machine-code blocks, "threading" them. An example of a Z80 inner interpreter is given on p. 29 of the book by Loeliger (1981), who comments (pps. 36-7) on the timing inefficiency of threading. The overhead timing loss of executing a "primitive" command (defined by a machine-code block) is 76 cycles, while a "secondary" command (defined by an intcode sequence, a *de facto* subroutine) costs 268 extra cycles for the transfer of control. This must be why the fastest version of Forth in the benchmark-comparison study of Gilbreath (1981) ran *six-fold slower* than the truly-compiled PL-I/80 or Whitesmith's C (all on a Z80 system).

The recent evolution of Forth-like HLLs, facilitated by newer CPUs with superlative addressing power, has altered the threading mechanism in order to minimize timing losses. In the Z8000 Forth system of Odette (*DDJ* No. 71) each primitive machine-code block is terminated by two machine-code instructions (4 bytes, 18 cycles) that cause a jump to the next primitive in the intcode sequence, so that a sequence of primitives "runs itself." Even if the next command is a secondary, transfer of control to its intcode sequence is expedited by prefacing that with a machine-code CALL COLON (4 bytes, 10 cycles, plus 35 cycles execution time for the COLON subroutine). Since the terminal return to the main intcode sequence by the SEMI command costs 25 cycles, the total timing penalty is 70 cycles. Relative to the Z80, this is a four-fold reduction in timing loss, at the cost of two more bytes added to each command.

The purpose of this note is to demonstrate that the Motorola 6809 can "thread" intcode with even greater efficiency, since its instruction set is rich in both direct and indirect indexed autoincrement codes. If the design makes use of its Y-index register as the equivalent of a CPU program counter, terminating each primitive machine-code block by JMP (,Y++) will cause a transfer of control to the next primitive in the sequence at a cost of only two bytes and nine cycles. These powerful addressing modes also allow what I call "in-program parametrization" of any intcode command. A simple example would be DROP (*n*), a command to "drop" *n* bytes from the data stack (using the U register as the stack pointer). The parentheses signal the text-interpreter to add the numeric *n* directly to the intcode sequence, following the execution address of DROP. The code for the parametrized DROP would be:

```
DROP   LDB ,Y+ (load binary n from
         program into B register)
       LEAU B,U (reset U stack pointer
         by adding n)
       JMP (,Y++) (jump to next
         intcode command)
```

This generalized command executes in 20 cycles for any value of *n* from 1 to 127. The "reading" of the parameter also advances the Y counter to point to the *next* command in the sequence. A similar parametrization in conventional Forth would be less efficient, being sourced as *n* DROP but intcoded as 5 (instead of 3) consecutive bytes: LIT address, *n*, DROP address. Execution of LIT would (a) pick up *n* from program, and (b) save it on the data stack. Execution of DROP would (c) retrieve *n* from the data stack, and (d) use it to reset the stack pointer. Steps (b) and (c) are clearly a waste of time. There is no need for a parameter that pertains exclusively to *one* command to visit the stack at all! The stack exists to save data generated by a command for use by a subsequent command; mere copying from the program is not generation in this sense.

In-program parametrization would likewise simplify the intcode and speed up execution of the DO . . . LOOP construct. I shall illustrate this for Loeliger's CDO . . . CLOOP since this variant is more efficient for loop indices from 0 to 255. For generality, I also make the loop-increment (*i*) explicit, since this adds only one byte to the intcode and only three cycles to loop timing, and it avoids cluttering up the dictionary with distinct CLOOP and +CLOOP commands. The source code is CDO (*n m i*) . . . CLOOP, where *n* and *m* are the initial and terminal indices of the loop. Here the 6809 S stack serves as the "return" stack.

```
CDO    LDA ,Y (pick up n from pro-
         gram into A register)
       LEAY 3,Y (increment Y to ad-
         dress of first in-loop command)
       PSHS Y,A (save address, then n,
         in S stack)
       JMP (,Y++) (jump to first
         in-loop command)
```

This executes in 26 cycles. Note that *n* is stacked because its value will be incremented during looping. The constants, *m* and *i*, can easily "read" by offsets using the stack-stored address.

```
CLOOP  LDX 1,S (pick up stored address
         into X index-register)
       LDD -2,X (pick up m and i into
         registers A and B)
       ADDB ,S (add i to the stack-
         stored n)
       STB ,S (save i+n, as new value
         of n)
       CMPA ,S (compare m to new n)
       BLS EXIT (if m is less or same,
         exit from loop)
       LEAY 2,X (move X plus 2 into
         Y)
       JMP (,X) (and jump back into
         loop)
EXIT   LEAS 3,S (reset S stack pointer
         before exit)
       JMP (,Y++) (and jump to post-
         loop command)
```

This more complex transfer of control costs 40 cycles for each run of the loop. Comparison with other systems would be tedious since timing information is never given, but I would be surprised if an equivalent Forth +CLOOP was as fast.

Conventional Forth allows use of different entry points into a complex machine-code routine, since all this requires is creation of a dictionary "name" that specifies the entry address. It's not clear whether multiple entry points into a complex intcode routine are also possible. In the design used by Odette, only *one* entry can exist into an intcode routine. Intcode-defined commands are, to the programmer, just "names" indistinguishable from those of primitives. If a 6809 system adopts the Odette model, the address of each secondary points to a 3-byte machine-code "leader" that is followed

by the intcode sequence. This is a JSR COLON (8 cycles) that pushes the address of the first intcode command in the sequence onto the S stack, not as a return address but for use by the COLON routine:

```
COLON LDX ,S (load new intcode
          address from stack into X
          register)
      STY ,S (save old intcode address
          in same location for return)
      LEAY 2,X (move X plus 2 into
          Y)
      JMP (,X) (and start running
          new intcode)
```

This takes 24 cycles, so the total entry timing loss is 32 cycles. Note that if the first two instructions were PULS X and PSHS Y the unneeded resettings of the stack pointer would add three cycles. However, resetting of the stack pointer is needed in coding the SEMI command, which terminates the secondary sequence and transfers control back to the original sequence:

```
SEMI  PULS Y (restore old intcode
          address from stack into Y)
      JMP (,Y++) (and return to
          running that intcode)
```

Since this takes 16 cycles, the total overhead of this form of high-level subroutine is 48 cycles – nearly four times longer than the machine-code JSR/RTS. It seems that the tradeoff for the conciseness of intcode is a several-fold slower execution time!

Entry to any execution address within a complex intcode sequence becomes possible if one creates a JSUB primitive, that must be followed by the "name" of the desired entry address (added to the sequence by the text-interpreter).

```
JSUB  LDX ,Y++ (load "name" address
          into X, set Y to return address)
      PSHS Y (save return address in S
          stack)
      LEAY 2,X (move X plus 2 into
          Y)
      JMP (,X) (and start running
          "name" intcode routine)
```

Execution time is 28 cycles, only four cycles less than the single-entry-point logic, and the "call" adds two bytes to the original intcode program. The SEMI return works as before. Intcode-defined commands must be "marked" so the text interpreter will recognize that they must be preceded by JSUB. Beyond the potential advantage of using segments of a complex intcode sequence, there is forced recognition by the programmer that use (and especially nesting) of JSUB degrades timing.

One of the intriguing possibilities of in-program parametrization is in-line insertion of a machine-code sequence within an intcode sequence. The shift would be enabled by an MCODE primitive, defined by a simple 2-byte, 3-cycle JMP ,Y, that will transfer control to the first machine-code byte. The machine-code sequence must be terminated by a JSR SEMI, which will transfer control to the following intcode sequence, at a 24-cycle cost, for a total cost of 5 bytes and 27 cycles. This would allow use of special-purpose operations without creating new dictionary entries for them, but of course makes the program system-dependent!

The extraordinary flexibility and user-extensibility of Forth-like HLLs is a mixed blessing, since it encourages customization and the proliferation of dialects almost to the level of one-man languages, the classic Tower of Babel effect. However, I am convinced that major conceptual changes (such as parametrization) are inevitable and will sooner or later be quietly adopted. They will provide at least some common core structure and enhance the competitiveness of this linguistic type. Standardization, to whatever extent possible, should be deferred until the wealth of ideas developed in existing dialects can be pooled into an optimized core. That is how a science develops, free of orthodoxy and sectarianism, always ameliorable by common agreement, never proprietary. It should not be necessary to create scores of fully-developed rival languages such as STOIC, URTH, SNAP, or the new RPL of Stryker (1982). That is why my controversial proposals for PARFOR (*DDJ* No. 52) were purely hypothetical guidelines for possible improvement of the linguistic type, and why I have here presented only a few of the command words I have written for my 6809 system. If they're not optimal, they ought to be replaced by whatever is.

DDJ

References

Gilbreath, J. "A High-Level Language Benchmark," *BYTE* 6(9):180, 1981.

Gordon, H. T. "PARFOR, A Theoretical Parametrized, Forth-like HLL," *Dr. Dobb's Journal* 6(2):8, 1981.

Loeliger, R. G. *Threaded Interpretive Languages*, Byte Books, Peterborough, New Hampshire, 1981.

Odette, L. "Z8000 Forth," *Dr. Dobb's Journal* 7(9):48, 1982.

Stryker, T. "BASIC, Forth, and RPL," *Micro* No.49:63, 1982.

A Common-Sense Guide to
Faster, Smaller BASIC

by Robert Irving

Robert Irving, 8637 Bothwell Road, Northridge, California 91324.

Editor's note: The advice in this article may be highly interpreter-dependent. As most DDJ *readers know, the inadequacies of a particular interpreter are neither universal nor necessarily long-lived. In the interest of readability and maintainability, many people deliberately sprinkle their code liberally with the very "inefficiencies" pointed out here, and later apply a mechanical compressor (several are commercially available, with varying degrees of thoroughness). Thus, two versions of each program can be kept – one for execution and one for documentation.*

In these days of increasing central processor speed and decreasing cost of memory, it is perhaps anachronistic to speak of optimizing speed and memory capabilities of your microcomputer. But assume for a minute that your home system isn't a Cray-1 with a couple of megabytes of RAM and a flock of 14" Winchesters. I remember just twenty years ago when the best military computer at sea (on the Polaris program) had a main memory cycle time of 80 milliseconds – yes, milliseconds, not microseconds. Running real-time navigation for even a submarine was touch and go. So we put eight read/write heads on one track of the drum(!), and got a 10 millisecond (Wow!) response.

Well, you're not that bad off. Let's assume your home system is a Z-80 and you've got 32K of RAM using a 12K BASIC. You've already got a system that is fast compared to the Polaris system, but you'd still like to get the most out of it. Here are some of the ways that you can do that. Obviously, you shouldn't expect that every one of these methods will improve speed and minimize memory usage. Some methods may do both, but most will do one or the other. In addition, some of the methods I will discuss will improve the accuracy of your results – when they can be applied.

First, let us consider speed enhancing techniques. Each individual application may appear trivial, but overall the effect is significant. Try some of the examples and you'll agree.

(1) Never put a REM statement in the middle of a set of FOR-NEXT statements.

Not this:

```
140 FOR J=1 TO 10
150 REM ** THIS IS A LOOP
160 A=A+J
170 NEXT J
```

But this:

```
140 REM ** THIS IS A LOOP
150 FOR J=1 TO 10
160 A=A+J
170 NEXT J
```

(In the first example, the REM statement would be read ten times; in the latter, only once. The more times the loop is iterated, the greater the time savings.)

(2) Avoid requiring the program to process a REM statement.

Not this:

```
270 GOSUB 7000
. . .
7000 REM **SUBROUTINE ARCTAN
7010 REM ** ANGLE IN RADIANS
7020 . . .
```

But this:

```
270 GOSUB 7000
. . .
6998 REM **SUBROUTINE ARCTAN
6999 REM ** ANGLE IN RADIANS
7000 . . .
```

(In the first example, two REM statements are read each time the subroutine is called. In the second example, no REM statements are ever read by the program; they show only in the listing.)

Not this:

```
500 REM ** USER INPUT
510 INPUT "NEXT CHOICE",U$
520 IF U$="END" THEN END
. . .
550 GOTO 500
```

But this:

```
499 REM ** USER INPUT
500 INPUT "NEXT CHOICE",U$
510 IF U$="END" THEN END
. . .
540 GOTO 500
```

(In the first example, the REM statement is read each time the program is returned to the user input by Line 550. In the second example, the REM statement is ready only once in the normal course of the program.)

(3) Use special symbols in lieu of blank lines to call attention to REM statements.

Not this:

```
160 NEXT J
170 REM
180 REM FUNCTION TO BE
    PLOTTED
190 REM
200 FND(X)=4*X*X+P1
```

But this:

```
160 NEXT J
170 REM ** FUNCTION TO BE
    PLOTTED
180 FND(X)=4*X*X+P1
```

(In the first example, three REM statements must be read by the program. In the second example, only one.)

(4) Keep REM statements as short as practicable.

Not this:

```
740 REM ** THIS IS A SUBROUTINE
    TO EXTRACT THE INVERSE
    TANGENT FUNCTION
```

But this:

```
740 REM ** SUBROUTINE ARCTAN
```

(In the first example, the program must read a much more lengthy statement, even though it is ignored in processing.)

(5) Minimize display length, amount of input, and input processing for interactive programs.

Not this:

```
210 INPUT "HOW MANY OBSERVA-
    TIONS DO YOU WANT TO
    ENTER",E2
```

But this:

```
210 INPUT "NUMBER OF
    SAMPLES",E2
```

Not this:

```
1430 INPUT "NEW GAME",U$
1440 IF LEFT$(U$,1)="Y"
     THEN 230
1450 IF LEFT$(U$,1)="N"
     THEN END
1460 GOTO 1430
```

But this:

```
1430 INPUT "NEW GAME (Y/N)",U$
1440 IF U$="Y" THEN 230
1450 IF U$="N" THEN END
1460 GOTO 1430
```

(The less time spent printing unneeded

displays and typing unneeded letters, the faster the operation goes.)

(6) Assign a symbolic name for frequently used constants.

Not this:

```
760 A1=3.141592654*R1*R1
770 A2=3.141592654*R2*R2
```

But this:

```
760 PI=3.141592654
770 A1=PI*R1*R1
780 A2=PI*R2*R2
```

(The interpreter reads a numerical constant character by character each time it is encountered. A constant stored as a symbolic is read character by character only once, and as a stored value subsequently.)

(7) Use simple variables (A, B, C, etc.) or nonsubscripted variables (A7, B4, C9, etc.) in lieu of subscripted variables [A(1), B(2,3), C(4), D(8,1), etc.] whenever possible. (Subscripted or array variables generally require more manipulations to store and retrieve, hence take longer.)

(8) Avoid the use of transcendental functions (trig, logs, exponentiation) whenever possible. For integral powers, use repeated multiplication.

Not this:

```
540 X=Y^2
```

But this:

```
540 X=Y*Y
```

(Trigonometric functions can be most time consuming. Raising a number to a power requires both LOG and EXP functions, lengthy operations, hence should be avoided for integer powers.)

Now let us turn our attention to techniques which reduce memory requirements. It is not surprising that some of the speed enhancing techniques also reduce memory. Specifically, (3) through (7) above reduce memory requirements. There is a version of (3) which is particularly wasteful of memory; one involving the use of multiple REM statements to outline the program title, author's name, copyright statement, etc., with rows and columns of asterisks or other symbols. Each of the symbols and spaces is individually stored in memory – and that takes a lot of memory for little real benefit.

Most of the prior examples are obvious as to the reason for the saving, but a couple require explanation. In (6), added memory is needed during operation to store the value of the constant symbol – but program memory is saved since multiple symbols and not multiple values are listed in program storage. In (7), the saving in using nonsubscripted variables lies in elimination of the storage for the array definition and its addresses. These overhead functions are not needed for individual variables.

There are other memory-saving techniques:

(9) Do not overdimension arrays. Many systems have a default value (e.g., 10 or 10,10) for arrays for which you have not written a DIM statement. It may save program storage to not dimension a 3 by 4 array, but when you accept the default 10 by 10, you waste an equivalent storage for a 7 by 6 array.

(10) For systems allowing multiple statements per line, use this capability to combine short statements. The colon or other separator symbol takes less storage than a new line number. Caution: Remember that most systems skip all statements to the right of an IF-THEN in a single line, unless the IF condition is TRUE.

(11) If your system allows, use the INPUT with PRINT capability.

Not this:

```
230 PRINT "LENGTH OF SIDE"
240 INPUT L1
```

But this:

```
230 INPUT "LENTH OF SIDE",L1
```

(The latter has limits, however. You cannot insert a variable inside the INPUT with PRINT format.)

(12) If your system allows variable string lengths, set STRING= to the lowest value that will encompass the strings you plan to use. This is especially important with arrays of strings since every cell in the array will reserve as many bytes as STRING= designates. A large array using the default value for STRING= (18 bytes in some systems) can eat up a lot of memory.

(13) Avoid isolated statements – ones which will never be executed because the program bypasses them. Every programmer has, at one time or another, introduced unused material into a program. A subroutine is inserted, but never called. A GOTO bypasses one or more lines of the program. These isolated statements waste memory and should be deleted. They most frequently result from inadequate review after a program has been "patched" or rewritten.

(14) Lastly, memory can be saved by eliminating spaces in the program. Caution: Be judicious in using this technique, even if you only have a small amount of memory. A statement like:

```
100FORI=ATOM:LETJORK=TRUE:
    NEXTI
```

could possibly blow a mental fuse!

So far there has been no discussion of accuracy. Only one of the techniques mentioned so far has an influence on accuracy. In (8) avoidance of transcendentals was discussed to enhance speed. In typical BASIC systems, the TRIG, LOG and EXP functions are typically one to three places less accurate than the arithmetic functions (+, –, *, and /). Thus the use of repeated multiplications in lieu of integer powers is not only faster but more accurate.

As for fractional powers and the TAN/ARCTAN functions, in most cases one uses them in BASIC and accepts the speed and accuracy penalties. If one must have the ultimate in accuracy for a transcendental, then evaluation of a truncated series expansion is probably the best route. However, a large penalty in speed is incurred, since the series methods are usually very slow.

From the foregoing discussion, we can conclude that there are some sure ways to approach maximum speed, minimum memory use, and maximum accuracy. Not all of these methods can be applied all the time. Some methods optimize speed and memory, others optimize speed and accuracy, but none optimize all three.

DDJ

GUEST ESSAY

A Fundamental Mistake in Compiler Design

by Edgar H. Fey, Jr.

Edgar H. Fey, Mobile Computers, 18 W. Calendar, La Grange, Illinois 60525.

I was stunned to read in *DDJ* that the MSDOS non-macro assembler requires 53,000 bytes of object code and the competing CP/M-86 assembler requires 27,000 bytes. I can only echo Dave Cortesi's comment, "... this kind of bloat is inexcusable."

Something is radically wrong. The authors of these assemblers are presumably competent programmers. The past records of their firms certainly indicate this. I am sure most of them are younger and smarter than I. The problem has to lie elsewhere.

In pondering this problem, I soon recognized that assemblers weren't the only culprits. In fact there is even more bloat in the typical compilers used today. There is also a fundamental reason for this bloating all down the line in typical software and the reason, I believe, is traceable to a fundamental mistake that compiler designers have been making for thirty years.

My conclusions are not based on a formal scientific proof. They are simply inferred from common sense observations. With that caveat I will first discuss bloating in assemblers and then turn to the real culprit, the conventional compiler.

Assemblers

The state of the art for the object code size of a good structured programming machine code assembler with all the IF...THEN, IF..ELSE...THEN, BEGIN ...UNTIL constructs one needs to automatically assemble branch instructions is on the order of 1500 to 2000 bytes. The size will vary depending on the instruction set of the processor but that is roughly what one should expect.

DDJ published an excellent example of a 6502 assembler by W. F. Ragsdale in the September 1981 issue. Ragsdale's assembler requires only 1300 bytes of object code. I have both a 6502 and an 8080 assembler, neither quite as elegant as Ragsdale's but each is about the same size and power as his.

Do not be fooled by the small size. These assemblers are every bit as powerful as the best bloated assemblers and much more convenient to use simply because it is practical to make them resident and part of your language. You can assemble and test a machine code program on the spot, interactively, without leaving you language. I do this routinely. This would not be practical if my assembler took 53Kb or even 27Kb. My machine simply isn't that big.

The argument that since memory is so cheap, the size of a program really doesn't matter is persuasive but false. You pay a lot more than the price of additional memory for poor programming. You pay in time, convenience of use, i.e. frustration, and in the ability to understand, change and modify the program. There is a world of difference in making a simple change to a 27Kb program as compared to a 2Kb program. Further, a little bloat in an application program is of a lot less concern than bloating in the assembler, compiler and other basic tools of computing.

The state-of-the-art assemblers I have mentioned were written in the Forth language but I am sure equally good assemblers can be written in Pascal or any other good language, if someone has not already done so.

Compilers

How big should a compiler be? I counted the space taken by the compiler functions in my Fig-Forth system. It uses only 350 bytes!

This is not one, but two orders of magnitude smaller than conventional compilers. Whether one likes the language or not is immaterial; the point is that if this is all it takes to perform the compiling functions of a powerful language (and Forth is a very powerful language), then what is wrong with conventional compilers? Where are they making their mistake?

If the compiler of your favorite language were reduced in size to 500 or 1000 bytes, you could make it a resident compiler which you could then use interactively. Wouldn't that be nice? You would then have room for the object code produced by the compiler and wouldn't need an operating system to continually swap things around.

By it being small, the compiler would be easy for you, as the user, to understand and change. You could make it extensible. Just think, if you then saw some feature you liked in some other language, you could extend your compiler to include that feature. And you could make the extension and test it on the spot, interactively, until the language was perfect for you. You wouldn't have to wait for the author of the compiler to make a set of changes that he thinks will be best for you. You make the changes that are best for you. Let him make the changes that are best for him. I call a small compiler instant computer freedom.

An extensible compiler has long been a goal of compiler writers. Most books that I have read on the subject begin with a syntax definition of languages. A particular syntax defining a language is then selected and most of the book explains how to implement the compiler based on the chosen syntax. This is really tough reading so I turn to the end of the book where I can usually find a statement to the effect: "... so that is how you write a compiler. Now it would have been nice if we could have made it extensible but that's simply not practical. All previous attempts in that direction have resulted in failure ...."

Folks, the problem of designing an extensible compiler does not lie in your techniques. They are super. The problem lies in Chapter 1:

If you start out with a complicated syntax for a language, you will end up with a large and complicated compiler.

It will then be too difficult for the user to extend.

One reason Charles Moore was able to write an extensible compiler for Forth is that he started out with a very simple syntax. It is so simple that it can be written in a few lines:

(1) A valid character is any member of the ASCII character set except the space and carriage return.

(2) A word is a string of ASCII characters delimited by an ASCII blank.

(3) A line is a sequence of words terminated by either the ASCII return, the established line length, or a 0.

(4) If a word in a line is

  (a) a previously defined word, it is executed.

  (b) not a previously defined word, then it must be either a number in the current base or an error.

    (1) If it is a number, it is placed on the stack.

    (2) If it is not a number, it is an error.

That's it. The language is expanded from this basic syntax. The compiler starts out very small and extensible and is kept extensible. Adding the syntax of the

usual structured programming constructs – IF...ELSE...THEN, BEGIN...UNTIL, DO loops, etc. – results in the final 350 byte Fig-Forth compiler which the user is free to expand at will.

One of the interesting products of Moore's approach is that by making the compiler small and extensible right from the very beginning, it appears that the language can be extended indefinitely. There does not appear to be an upper bound to the complexity of the language. If a user creates a syntax that appears to box him in, he should always be able to leave a path for continued expansion in some other direction.

There may be other simple syntaxes that will lead to small, powerful, efficient, interactive, extensible compilers, but the syntax used by Moore is known to work. It also may turn out that all such simple syntaxes yielding extensible compilers result in languages which have characteristics quite similar to Forth. Personally I think that would be just great, but I am sure there will be some initial objections simply because that is not the way things have been done for the last thirty years.

For example, the languages may consist of almost all verbs. Although you can put them in if you wish, there are no expository statements in Forth. Every word produces an action. The traditionalist who has been taught otherwise may say, "That's terrible – how can you express yourself in a gutteral language that doesn't have elegant expository statements?"

This is a loaded question that really has three answers:

(1) Of course you realize that you can get the computing job done. That's what verbs are all about.

(2) Do not confuse program documentation with computing. They are two separate functions. The user should not be forced to document when he feels it to be unnecessary and all he wants to do is compute. Of course it should be made convenient for the user to document to his heart's content if that is what he wishes to do.

(3) The real answer to the gutteral language charge is that with an extensible language, the user has complete freedom of expression simply because he, not the compiler writer, *controls* the syntax of the language. The user is in a much better position to decide what syntax he needs or would like in a particular application to express his ideas properly. He can try out a particular syntax and test it immediately. If it is unsatisfactory, he can then change it. The guiding principle is that since the bloat in compilers is related to the syntax, it is better to let the user control the syntax. The bloat will then tend to be at the top in the application and not at the bottom where it hurts.

All this talk about changing the compiler and changing the syntax must seem quite vague and mysterious to those readers who have not had the opportunity to actually do this sort of thing. It isn't. Let me try to explain with a practical example.

Say I have a file system and I wish to define individual record components in a file and give them unique names. I also want to fetch and store the record components without having to remember their data type or their offset location in a record. Let's call the fetch and store operators V@ and V!. If the name we assign to a record component is xxxx, then we would like to be able to use the following syntax for fetching and storing data for xxxx in the current record:

```
xxxx V@  to fetch the component
xxxx V!  to store the component
```

If the data type of xxxx is numerical, we will use the convention that V@ puts the fetched number on the parameter stack. If its data type is string, V@ will place the fetched string on the string stack. V! will be similarly designed. What we are trying to do is design universal fetch and store operators that receive information on the component's data type and location directly from the name of the component.

Once a protocol is set up as to how the type and offset information is to be transferred, the design of V@ and V! is straightforward. For example, xxxx could leave two numbers on the parameter stack to be picked up and used by the operators.

The compiler change comes in when we try to define xxxx. We could use the ordinary compiler to define xxxx so that when it is executed it leaves the proper numbers on the stack. But then each time we encountered a new component, we would have to tailor its definition to its data type and its offset. Sooner or later we would foul up and make a mistake. The compiler should take care of that sort of thing. So we change the compiler.

For this situation I added several new defining words to the compiler: INTEGER", BYTE", DINTEGER", and STRING", They are used to define record components in the current file, allocate space in the file records, and, when the newly defined name is invoked, pass the proper information to the fetch or store operator. It was not very difficult to do. It was all done in one session and the system was tested interactively without leaving the language.

Now when I execute

```
STRING" PARTNAME
```

I define a record component named PARTNAME in the current file. If, for example, I place the string GIZMO ONE on the string stack and then execute

```
PARTNAME V!
```

the string GIZMO ONE will be placed on the disk in the current record as the value of PARTNAME. Similarly then executing

```
PARTNAME V@
```

will fetch the string GIZMO ONE back to the string stack.

We could, of course, get fancier and have the new compilers also link the new record components to components in other files, but as you have seen, the ability to decide on a syntax change and then implement that syntax by modifying the compiler has obvious practical uses. It is a very creative form of endeavor that pays compound interest by making all subsequent computing easier.

Comments on "Fifth Generation Computers"

Editor's note: The good Doctor enjoys seeing dialogue among the readers. Recently we received the following piece in response to the guest essay "Fifth Generation Computers" by Richard Grigonis which was published in the December 1982 issue of DDJ. *It provides some interesting counterpoints which seem likely to interest readers, and perhaps to generate more comment on the issues discussed.*

I admit that the cost-performance ratio of, say, putting an IBM 370 on a Motorola 68000 – a plan which IBM is seriously considering – should spell doom for at least the superminicomputer manufacturers. But Richard Grigonis, in his essay "Fifth Generation Computers" (*DDJ* No. 74), is not just talking about "microsuperminis" or even "micromainframes." He asks us to believe that a supercomputer, 64-bit processor running at 110 megahertz (which is essentially a Cray 2S) can be put on a single chip.

It will take a while for VLSI technology to catch up to mainframes in terms of performance and fault tolerance – even mainframes have fault tolerance problems. Redundant circuitry must always be built to take care of such things as processors, main memory, and recovery from faults in the I/O system. Grigonis' secondary 32-bit I/O processor, probably used for maintaining the graphics, would itself require error detecting circuitry.

The 64-bit main microprocessor would have to be so small (in order to achieve 110 megahertz) that Heisenberg's Uncertainty Principle would come into play, and the positions of the electrons in the signal pathways of the processor would become "blurred," leading to strange field effect in neighboring signal pathways. What all this means in that 90% of his processor would have to consist of error-correcting circuitry.

Granted, slower 64-bit processors will probably be in use as early as 1986. I am surprised, however, at the preciseness of his prediction that processors of *110* megahertz will have been developed by the year 1992. It would have been more realistic to suggest a barrier of 100 megahertz instead.

The gigantic 4-megabyte EEPROMS Grigonis postulates may in fact be developed by 1992, but probably for mainframes, not micros. Even today the most advanced micro technology only provides the main memory chips used in mainframes. The microprocessor arm of this technology has been incorporated in mainframes only in the form of controllers and intelligent terminals. Microprocessor technology cannot be applied to mainframe processors because of logic circuit "randomness" problems and a general lack of performance, at least until recently. More gates than those projected for future microprocessors would be required.

Also, mainframe processor gates have switching times of about one nanosecond, as opposed to the three- to five-nanosecond switching times of the best microprocessors such as the eight-megahertz Z80-H and the 16-megahertz version of the Motorola 68000. The Grigonis 64/110 processor would probably have a switching time of about half a nanosecond – which, coincidentally, pushes silicon to its *theoretical* physical limits. I don't think a single chip made of silicon can handle the gate densities required for the operation of a processor with the Grigonis specifications.

Those 10,000- to 15,000-megabyte optical disk drives (10 inches in diameter) also give me pause. A conventional read-only videodisk can easily yield about 2,250 megabytes of storage (333,333 bits x 54,000 tracks), so Grigonis is talking about a five- to seven-fold density increase, *with* the ability to write to the disk. While I wouldn't say that's impossible, given the unknown technological developments that will occur over the next ten years, I think about 3,500 megabytes on a 14-inch disk sounds more realistic, at least as far as small systems are concerned.

Grigonis also mentions artificial intelligence programs that could require 32 megabytes of memory with a virtual memory system. Since the internal memory working set size of any virtual memory system is usually half the size of the program, he evidently envisions AI programs of 60 or 70 megabytes in length. These would be several times the size of the standard business applications packages that his AI program would presumably replace.

The problem of software quality assurance also rears its ugly head. In an actor-based, artificial-intelligence driven, customized language generator as described by Grigonis, how does one deal with program validation if the language has never before existed and the programmer is unfamiliar with it, yet is fully understood by the "machine" (meaning the actor-based AI program) that developed it? Can we be sure that the AI program really "understands" its own creation? Normally, locating errors in code that is written in a well-known language such as COBOL can take up to half of one's time – what about the unenviable position of having to locate errors in a program written in a language totally foreign to everyone?

Besides, it is my strong feeling that even with artificial intelligence, 64-bit microprocessors, and many megabytes of memory, the supermicros of 1992 will all serve merely as intelligent terminals to even more powerful mainframes, especially in the videotex environment.

Still, after years of looking at *DDJ's* countless pages of software listings in weird languages, it was stimulating to read "Fifth Generation Computers," and I would be curious to see Grigonis' speculations on the workstation environments that will have to be developed to contain the operator as well as the advanced hardware he describes.

DDJ

by Michael J. Doherty

Michael J. Doherty, 334 South Maple Avenue, Glen Rock, New Jersey 07452.

Circle **no. 65** on reader service card.

CP/M EXCHANGE

by Gene Head

Last month this column carried the source listing for a sorted and sized directory listing program, DIR.ASM. One of the more obscure sections of that code dealt with the Disk Parameter Block and the computation of space available on a partially filled disk.

Frankly, I don't understand as much as I would like to about disk tracks, sectors, records and the like. Bob Blum has contributed two articles that he says will clear up some of the foggy parts about basic disk I/O. I'm very encouraged by Bob's contribution to this month's and next month's column for three reasons.

First, Bob is a perfect example of readers sharing their experience with the rest of us; *Dr. Dobb's* specializes in this. Second, Bob wrote the article and sent it to me via modem; our computers are helping us communicate effectively! Finally, I hope other readers will see how helpful *their* input is and send in their ideas.

Perhaps the mysteries surrounding the CBIOS to BDOS interface have discouraged you from upgrading to a higher performance disk system or from making a few of those changes that could make your system more livable. In this, the first of a two-part series, we will review the major aspects of the most popular disk drives in use today and the data allocation methods used. This will set the stage for next month's article where the disk interface portion of the CBIOS will be covered in detail. At the conclusion, I hope you will feel more comfortable with CBIOS's inner workings and consider it a friend rather than a foe.

3740 Format

Several years ago, IBM introduced the 3740 data entry system which used 8" floppy disks for external storage. As is characteristic of the computer industry in general when IBM introduces a new product, it is soon copied and many times becomes a new standard. This is the case with the 3740 disk format. It was the *first* format to be implemented under *CP/M* and continues to be the standard today. For that reason it will be used here as the basis for example until the discussion turns to the newer, more sophisticated disk drives.

The 3740 configuration is implemented using a single-sided, single-density, soft-sectored disk drive. Single-sided means that only one side of the media is used for recording by a single read/write (R/W) head. Referring to Figure 1 (below), each track of recording area is divided into data areas called sectors. Separation of consecutive sectors is accomplished through control information recorded along with the data portion of the sector. This type of sectoring is referred to as soft-sectored format. To reach any one of the sectors, all we need to know is the track and sector numbers. From there the disk controller and CBIOS logic direct the disk drive to step to the appropriate track and await the arrival of the desired sector. Sector skewing is employed to increase the speed at which sectors can be sequentially accessed. This technique offsets consecutive sectors from each other by a count of 6. Figure 4 (page 81) lists the standard skew table used in the 3740 standard. The skew table is used to map logical sectors to their physical location on disk. For example, your program wants to read logical sector 2. The BDOS first calls a translation routine which uses the logical sector number 2 as an index into the skew table. In position 2 is 7, offset from 1 by 6, which is used for the actual read operation. The rate at which consecutive sectors can be sequentially accessed is greatly improved by skewing because most of the time it is possible to process the data just accessed and prepare for the next operation before the next sequential sector passes under the R/W head. This makes it possible to access up to two sectors per disk revolution. Without sector skewing, the best possible data rate would be one sector per revolution of the disk.

CP/M 3740

In CP/M 3740 single-density format there are 26 128-byte sectors per track and a total of 77 tracks per disk. Sectors are numbered from 1 to 26 and tracks from 0 to 76. The first two tracks are reserved for system use. The boot loader occupies track 0 sector 1 while the remaining sectors on track 0 and 1 are used to hold the operating system. All remaining tracks are available for directory and data area usage.

One of CP/M's major advantages is the ability to dynamically allocate disk space during program execution. To accomplish this, allocation groups are used. Each group contains eight consecutive sectors or 1024 bytes and is the basic unit of disk space which can be allocated at any one time. Control of the allocation and deallocation of groups during execution of a program requires that the current status of each group be available at all times. Memory-resident tables are used for this purpose. You have probably noticed that when logging in a new drive or warm starting, quite some time is spent in

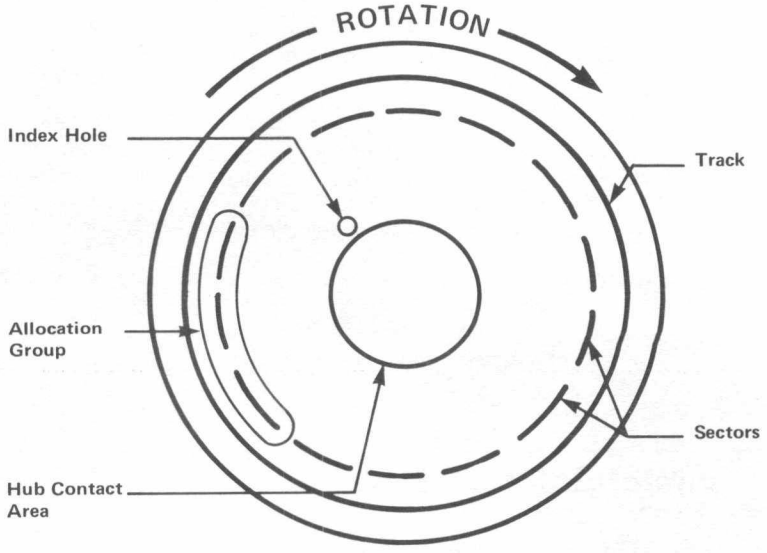


FIGURE 1.

disk activity. During this time CP/M is reading the entire disk directory and constructing a bit map in RAM. The bit map is appropriately named because each bit in the map represents the status, in-use or available, of one allocation group. The purpose of allocation groups is to reduce the memory requirements of the bit map and to prevent excessive file fragmentation. For example, if each allocation group were one sector in length, the bit map would contain 1944 bits or slightly over 243 bytes. Since the maximum capacity of the 3740 system is 243 allocation groups, only 31 bytes are needed to contain the bit map for each active drive. This is a large memory saving, not to mention the reduction in head movement necessary to process 1024 bytes of data. To find an allocation group in which to write, CP/M searches the bit map sequentially from the beginning until a zero bit is found which indicates availability. During the search, each bit that is passed over is counted. The resulting count is used to identify the allocation group and is used in subsequent operations to determine the track and sector numbers. When returning allocation groups to the system, the directory entries of the appropriate file are read and the allocation group numbers are extracted from the FCB and are used as an index into the bit map to appropriately adjust it.

Double-Sided Drives

As microcomputers found more common use in commercial data processing, more capacity on the same size media created the need and subsequent introduction of the double-sided drive. In this configuration there is one stationary and one movable R/W head as in the single-sided drive. This type of drive still gives engineers fits because of unacceptable wear patterns, media warping when both heads were engaged and gravity problems since the lower head must be held in place against gravity. This also presented some new problems for the driver software and the disk controller because it was now necessary to command the drive to read or write from a particular head in addition to track and sector. One method of adapting to this new drive was to map one logical single-sided disk to each side of the two-sided disk. This achieves the desired result of twice the storage capacity but has one inherent disadvantage – excessive head movement which slows data transfer, sometimes to an unacceptable limit. The common solution was to logically view the drive as having twice as many tracks as it really had. Referring to Figure 2, it is common on a 40-track, double-sided drive to assign all tracks on size zero even numbers between 0 and 78 while side one's tracks are assigned odd numbers between 1 and 79. Since the disk controller will only recognize tracks 0 through 39, the CBIOS has to translate logical tracks to physical tracks and set the appropriate head selection line. This calculation is not too difficult. Referring to Figure 3, divide the logical track number by two. The quotient is the physical track number and the remainder is used to select the appropriate head. From this we can see that both tracks 0 and 1 are under the R/W heads at the same time and can be visualized as a cylinder. This technique provides faster data transfer because no mechanical movement is necessary to process two complete tracks of data.

Certain manufacturers weren't satisfied with only double the capacity, so double-density recording methods were developed which provided approximately twice as much data content in each of the sectors and twice the transfer rate. Today, even more data is being packed onto each disk by increasing the number of tracks and using group encoding. It's fairly common to find 640K bytes of data being stored onto one 5¼" floppy disk. With motor speed control, it is possible to store 512K bytes on a single-sided disk system as on the Victor 9000. The reward for greatest capacity on a 5¼" floppy is held by a British firm, Rair, which offers over 800K on one disk by double-sided, quad-density formatting.

Along came hard disks and still further increased storage capacity. It is not uncommon to have 200 tracks and six or eight recording surfaces provided by three or four platters on hard disks. Data transfer rates are also considerably faster because rotation speeds of approximately 3000 to 3600 RPM are common. With each movement of the R/W heads, more data are available because the cylinder has grown to multiple surfaces.

There is much more to cover in this track and sector discussion. If you have further questions, send me a SASE and I will return to you a list of reference materials which will explain these subjects in greater detail. Next month we will get straight to the heart of the interface tables and their relationship to the system.

DDJ

(Figures 2, 3, and 4 on next page.)

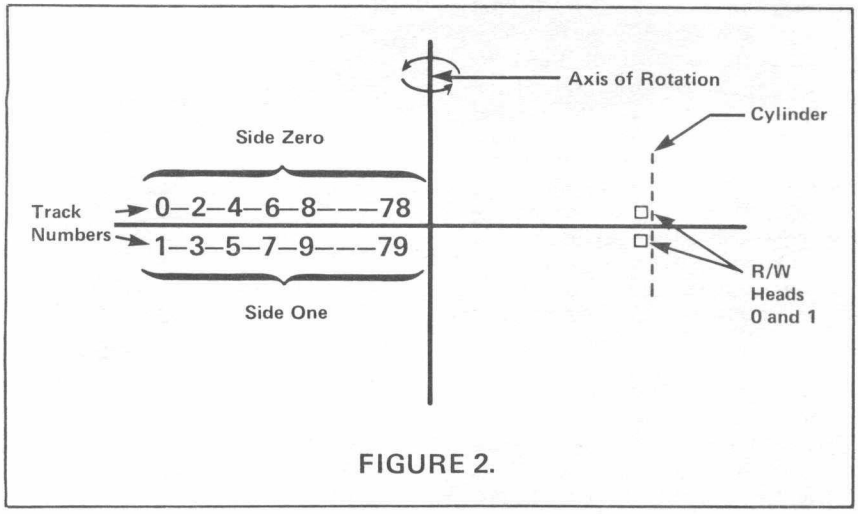


FIGURE 2.

```
          20 (PHYSICAL TRACK)
        ____________________
2   |   41 (LOGICAL TRACK)
    |   40
    |   --
        1 – SIDE
```

FIGURE 3.

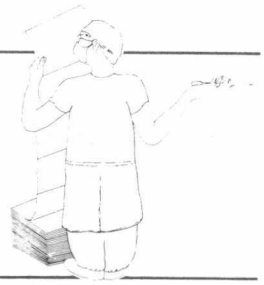
1, 7, 13, 19, 25, 5, 11, 17, 23, 3, 9, 15, 21, 2,
8, 14, 20, 26, 6, 12, 18, 24, 4, 10, 16, 22

FIGURE 4.

	Format	Code
1.	8" Single-Sided/Single-Density	8SS/SD
2.	8" Double-Sided/Single-Density	8DS/SD
3.	8" Single-Sided/Double-Density	8SS/DD
4.	8" Double-Sided/Double-Density	8DS/DD
5.	5¼" Single-Sided/Single-Density	5SS/SD
6.	5¼" Single-Sided/Double-Density	5SS/DD
7.	5¼" Double-Sided/Single-Density	5DS/SD
8.	5¼" Double-Sided/Double-Density	5DS/DD
9.	5¼" Double-Sided/Quad-Density	5 DS/QD
10.	5¼" Single-Sided/Double-Density/Variable Speed	5 SS/DD/V
11.	5¼" Double-Sided/Double-Density/Variable Speed	5DS/DD/V

Table 1.

DR. DOBB'S CLINIC



by D. E. Cortesi

The Eternal Calendar

Calendar algorithms are an unfailing source of fun and agony for programmers. Early in this column's history, we proposed some calendar code. We also worked on some problems in BASIC coding whose solutions revolved around using the arithmetic value of relational expressions. Brian Moore of Oakland, CA, was going through his back issues of *DDJ* and was inspired by those old problems to propose a calendar algorithm for BASIC which uses the arithmetic value of relations.

Moore's algorithm takes a day-of-the-year number as its input, and a boolean value that is "true" if this is a leap year. In one big expression, it biases or offsets the day number to the value it would have if all months were 31 days long. The month number and the day-of-the-month number are obtained from that value by dividing by 31. Our version of Moore's code (Listing 1, below) is for the more common BASICs, in which "true" is represented as negative one and "false" as zero. Some BASICs represent "true" as positive one; if that is true of yours, change all minus signs in line 1040 to plus, and vice versa.

The Productive Erase

Aubrey Hutchison, of Pompano Beach, FL, sends us an idea that is (we think) eccentrically brilliant. First, some background. CP/M 2.2 supports the notion of a user number, a number in 0..15 that is an implicit part of all filenames on a disk. There is a current user number (set with the USER command) which qualifies all searches of the directory, so that the only files you can see are those that were created under the current user number. It's a nice idea – it allows the directory of a large disk to be partitioned into as many as 16 sub-directories – but it wasn't carried out thoroughly enough. Later versions of CP/M (CP/M 3, Concurrent CP/M, MP/M 2, and MP/M 86) make the user number more useful.

One problem with the user number (as implemented in CP/M 2.2) is that there is no simple way to move files from one user number to another. You can do it with PIP; its [G] option will cause it to search for the source file(s) under a different user number than the current one. But since the only commands you can use are those stored under the current number, you have to have a copy of PIP under every user number you'll use. And if you want a file to appear under two user numbers, you have to have two copies of it, one stored under each number.

Well, Hutchison was pondering these matters, and thinking about the fact that the user number is stored in the first byte of directory entry as a value from 00h to 0Fh, when he had a satori of sorts. He flashed on the fact that, when the BDOS erases a file, it simply stores E5h in the first byte of the file's directory entry. So changing the user number of a file and erasing it are identical operations, namely, storing something in the first byte of the directory entry. In order to change the user code of a file, all that is needed is to persuade the BDOS to use the user number instead of E5h for erasure, and then to erase the file.

Hutchison went looking for the magic constant of E5h in the BDOS, and he found it. Just 655h bytes below the warm-start entry point of your BIOS (whose address is in location 0001h) there is an instruction, "MVI M,E5h." If that is changed to, say, "MVI M,03h" and the BDOS is called to erase a file, the file won't be erased. It'll be given user number 3 instead. Of course, the BDOS is refreshed from disk whenever a warm start is done, so the zap has to be made on the fly. You can't expect to do it with DDT, because the BDOS will be refreshed when the DDT ends.

But you could write a GO-USER command that takes a user number and a file-spec, patches the BDOS on the fly, erases the filespec, and ends. Before the command ends, it should un-patch the BDOS. That's because there are a few times when the BDOS is not reloaded from disk on a warm start – like whenever XSUB is running.

A Pascal Standard At Last!

Members of the IEEE Computer Society got gladsome news in the mail just before Christmas, in the form of an ad for the book *American National Standard Pascal*. According to the ad, "IEEE Standard Pascal has completed its IEEE approval and is undergoing final ANSI approval at this time . . . (it) provides an unambiguous and machine independent definition of the language." The cloth-bound book is the official standards document; it costs $17.95 to non-members, $16.95 to members of the Computer Society, and should be shipped in January. Order from IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720.

We know at least one software outfit that has made frequent public promises to bring its compiler into conformance with the standard language "just as soon as there is an approved, not a draft, standard." Don't hold your breath; they'll probably claim they really meant an *international* standard, not just an American one.

Dempsey's Dilemma

J. Dempsey, of Seattle, WA, has a sticky problem, one that most computer makers would prefer not to think about. Dempsey says, "I do writing in a technical field (linguistics) where I need to use a variety of special symbols. As far as possible, these should have equal status with the regular alphanumerics. I've seen articles on how to generate such symbols

Listing 1.

```
1000 REM GIVEN: DAY IN 1..366, A DAY OF THE YEAR, AND
1010 REM        LEAP = TRUE (-1) OR FALSE (0),
1020 REM RETURN MONTH IN 1..12, DATE IN 1..31
1040 DI = DAY - 1
          - (DAY > (59-LEAP)) * (3+LEAP)
          - (DAY > (120-LEAP))
          - (DAY > (181-LEAP))
          - (DAY > (273-LEAP))
          - (DAY > (334-LEAP))
1050 MONTH = INT(DI/31)+1
1060 DATE  = (DI MOD 31)+1
1070 RETURN
1080 REM GIVEN YEAR IN 0..9999, RETURN LEAP TRUE OR FALSE
1090 LEAP = ((YEAR MOD 4)=0) * ((YEAR MOD 400)<>0)
1100 RETURN
```

on an IBM PC, Apple, etc. but always in a graphics, not a text, mode. [I assume that such characters] are therefore in poorer resolution than ordinary ones, clumsy to use from the keyboard, and presumably useless for word-processing or data bases. What do I do?"

What, indeed? Before you rush to answer, take a close look at one of the examples Dempsey sent along (see box, this page). It's all very well to note that the Atari, the PC, the Victor 9000, etc., all have screen character sets that can be partly or completely redesigned by the user; getting the symbols onto the screen is only a tenth of the battle.

Let's call symbols like the ones Dempsey works with *specialist* symbols: symbols not in the standard ASCII set, but which are commonplace in some specialist's field – linguistics in this case. Law, medicine, mathematics, indeed every profession, makes use of its own set of specialist symbols. The International Phonetic Alphabet is one set of specialist symbols that it would be nice to have when working with, or writing about, the Votrax speech synthesizer. Choreographers have an "alphabet" of dance movements; at one time IBM offered a Selectric typeball for it. Someone editing a dictionary would like to have a set of diacritical marks. And so on.

With many personal computers, we can put a set of specialist symbols on the screen, but only by *replacing* some part of the machine's native font. The specialist characters are folded into the "letter space" of 0..255, and so are *ambiguous* when stored in RAM or a file. A file containing them might look right when displayed on the screen (provided the right user-defined text font is loaded), but will those byte values be processed correctly by a sort? And how can they be printed? A fundamental problem is that the 8-bit symbol space is too small to accommodate more than one alphabet. This doesn't hurt when the problem is to accommodate a foreign language (although there is a problem when the foreign alphabet has a large number of symbols, as with some oriental languages). But specialist symbols are an extension of the standard alphabet, not a replacement for it.

There's a mechanical problem, too: how do you arrange a keyboard to allow for a large vocabulary of specialist characters? The usual answer is, as before, folding – some keys are preempted to stand for special codes. One of the IBM PC's best features is its ALT-shifted key values. When we first saw it, we thought it was useless, but the longer we work with it, the more uses we find for that third shift-mode.

At any rate, has any reader got either practical hints for Dempsey on what system and software to look at, or more general comments on the whole problem of specialist symbols?

Backward with the PC . . .

The IBM PC does something very odd if you write a Backspace code (08h) from BASIC. If you write a Backspace when the cursor is at the left margin, nothing happens (which is just as it should be). But if you write a Backspace when the cursor is away from the margin, you get, not a leftward movement of the cursor, but a funny little graphics symbol, a small reverse-video diamond. Try this to see it:

```
PRINT ">"+CHR$(8)
```

Beyond the irritation of not having a Backspace that backspaces, we find this puzzling because we pored over the BIOS listing in the Technical Reference manual and we can't see where the bug is. Maybe it's in BASIC. Furthermore, the symbol displayed is not any of the ones documented in any PC manual that we can find, nor is it a reverse-video version of one of them. Can anyone explain these things?

. . . Also Up, Down, and Around . . .

If you are using CP/M-86 on a PC, you will find that the BDOS has a rudimentary form of terminal emulation built into it, so that the PC's display can be programmed as if it were a terminal with an addressable cursor. The BDOS (not the PC's firmware) supports escape sequences to set the cursor location and to access various other features.

If you are using MSDOS, you don't have this option. Cursor addressing becomes a matter of some rather advanced PEEKS and POKES, or their equivalent in whatever language you are using. One thing that might help is to note that the ASCII control characters FS, GS, RS, and US (1Ch, 1Dh, 1Eh, and 1Fh) are defined by BASIC as moving the cursor right, left, up, and down, respectively. However, this appears to be a BASIC function, not something built into the ROM BIOS or the CRT controller chip. From assembly language, you can call the ROM BIOS directly; what you do from Pascal or COBOL, we don't know. Do you?

. . . And BASICally Slower.

Bob Pirko, of New York, took us up on our recent challenge; he has explored the contents of IBM BASIC and demonstrated to his own satisfaction and ours that it is definitely a mechanical translation from 8080 code. He writes as follows:

"I own an IBM PC, and like you, I found IBM BASIC to be slower than I had expected. Having traced the execution of a few statements in BASIC, I can offer some reasons for lack of speed.

"Much of the code is undoubtedly a mechanical translation of 8080 code. Attached are three samples (see Listing 2, page 85) which list the code from the IBM PC's ROM, the equivalent 8080 code, and more efficient 8088 code. Table 1 (page 84) gives execution times for each of these routines.

"The IBM ROM routines are two to six times slower than they could be, and approximately twice as slow as comparable 8080 code. While such cases demonstrate the perils of mechanical translation, most of the code in IBM BASIC is not

An example of "specialist symbols."

jaķ nafar dehaːti bema ʃaːhr	*dar baːzaːr aʃo*
A peasant came to the city.	He was going along in the bazaar.
berasaː be dokkuːne yannaːti	*ʃuːriniːhaːje rangi rangi*
He reached the shop of a confectioner.	Confectionery of different kinds
dokkun darʃun tʃiː bu	*iːn ʃiːriːnforuːʃ aːhe ve*
was set out in the shop.	This confectioner was sitting and
angaʃt merde dehaːtiː xijaːlɛʃ ķa yannaːd kuːrɛ	
looking. The peasant thought that the confectioner was blind.	
dzaldi beʃo	*merde dehaːtiː do taː ungguːʃt dzele*
He went (forward) quickly.	The peasant held his two fingers before
tʃamʃɛʃ daː	*yannaːtɛʃ bɛʃva tʃera iːn*
his (the confectioner's) eyes.	The confectioner said to him, "Why did
kaːrɛt beķe	*merde dehaːti bɛʃva mon xijaːlem ķe to kuːriː*
you do this?"	The peasant said, "I thought you were blind."
mon kuːr nehaː avːnon	*dehaːti bɛʃva ajar to avːni tʃeraː to*
"I am not blind; I see."	The peasant said, "If you see why
ʃiːriːnijaː naxeri.	
do you not eat the confectionery?"	

Table 1.

Sample Execution Times In Micro-Seconds			
	IBM ROM *	Equivalent 8080 **	Efficient 8088 *
1. Add AL to BX	8.9	4.0	2.6
2. Immediate minus BX	10.5	6.0	5.3
3. String Move(20 bytes)	409.6	188.0	72.9

* Based on time test run on an IBM PC.
** Calculated for 5 Mhz. 8085.

quite so bad. In fact, some of it has been optimized to use the superior features of the 8088's instruction set. Still, most routines take longer than similar 8080 code executing on a 5MHZ 8085 or even a 4MHZ Z80.

"Generally, the 8088 will execute fewer instructions per second than an 8085 or a Z80. This may seem surprising, but shouldn't be. The rate at which instructions are executed is only one of the factors which determine speed. Of equal importance is the quantity of useful work done by each instruction. Happily, the 8088 repertoire contains a large number of operations which do as much work as two or more 8080 instructions. It is only by effective use of these instructions that an 8088 can reach its true potential. Translated code is slow because it consists of just those 8088 instructions that are functionally equivalent to 8080 instructions.

"The 8080 has many one-byte instructions that execute in one memory access cycle. The equivalent 8088 code requires two bytes and often takes two memory cycles (eight clock cycles) to execute. Intel manuals list shorter times, but they assume that the instruction has been pre-fetched and is waiting in the internal queue when needed. This is often untrue, especially for the kind of code produced by a translator program.

"In summary, even if Microsoft had done a better job at optimizing the translated code, the performance would still have been below expectations. Really efficient code would have required a complete rewrite of BASIC. This would be an ideal project for someone with a couple of years available and nothing better to do."

DDJ

Listing 2.

Example 1 -- Add 8 bit value in AL to BX

IBM ROM Code			Equivalent 8080 code		Efficient 8088 code	
ROM Address	ROM Code					
F600:1893	ADD	AL,BL	ADD	L	CBW	
F600:1895	MOV	BL,AL	MOV	L,A	ADD	BX,AX
F600:1897	ADC	AL,BH	ADC	H		
F600:1899	SUB	AL,BL	SUB	L		
F600:189B	MOV	BH,AL	MOV	H,A		

This code is executed whenever one of the four arithmetic operators is specified. It is used to compute the address of the appropriate arithmetic routine. The value in AL is always less than 7FH so CBW gives the correct result. The efficient code does not preserve the contents of AH as does the ROM code. This is no problem because the subsequent code does not expect any value to be passed in AH. Indeed, it would be surprising if it did since translated code doesn't know that the AH register even exists.

Example 2 -- Subtract BX from an immediate value (0FF26H)

IBM ROM Code			Equivalent 8080 code		Efficient 8088 code	
ROM Address	ROM Code					
F600:2CE0	MOV	AL,26H	MVI	A,26H	NEG	BX
F600:2CE2	SUB	AL,BL	SUB	L	SUB	BX,0DAH
F600:2CE4	MOV	BL,AL	MOV	L,A		
F600:2CE6	MOV	AL,0FFH	MVI	A,0FFH		
F600:2CE8	SBB	AL,BH	SBB	H		
F600:2CEA	MOV	BH,AL	MOV	H,A		

This is executed once for each variable in a BASIC statement. I think this routine checks whether the stack has overrun its limits. I used SUB BX,0DAH rather than the more natural ADD BX,0FF26H in order to set the flags the same as the original code.

Example 3 -- String move

IBM ROM Code			Equivalent 8080 code			Efficient 8088 code	
ROM Address	ROM Code						
F600:2895	INC	BL		INR	L		
F600:2897	DEC	BL	MOVEIT:	DCR	L	MOV	SI,CX
F600:2899	JNZ	289CH				MOV	DI,DX
F600:289B	RET			RZ		MOV	CL,BL
F600:289C	MOV	SI,CX				XOR	CH,CH
F600:289E	LODSB			LDAX	B	REP	MOVSB
F600:289F	MOV	DI,DX				MOV	DX,DI
F600:28A1	STOSB			STAX	D	RET	
F600:28A2	INC	CX		INX	B		
F600:28A3	INC	DX		INX	D		
F600:28A4	JMP	2897H		JMP	MOVEIT		

This routine is used by BASIC to move string variables. It shows how bad mechanical translation can be. The 8080 uses BC and DE as memory pointers. The Intel standard is to equate BC and DE with CX and DX respectively, and the ROM obeys this convention. Since CX and DX can't be used as memory pointers the ROM must add instructions to move the addresses to SI and DI.

OF INTEREST

by Michael Wiesenberg

Move Over Ada, Here Comes Beb

The Department of Offense has announced Beb, their new programming language named after Mary Shelley's maid's daughter, B'eb, now conceded to have been the world's first computer programmer. B'eb is reliably documented as having programmed an idiot savant to balance her checkbook in 1829, three years before Lord Byron's daughter, Ada, even met Charles Babbage (and, in fact, several years before checkbooks were even in the hands of the general public). Beb uses the IF . . . THEN . . .ELSE . . .BUT ON THE OTHER HAND construct. A RANDOM GOTO statement randomly jumps to any program line; its EXTERNAL option causes a jump to any random memory location, including some not accessible to the processor. Beb saves program space by writing all data to code space. Not only does Beb have no I/O routines, it does not permit them to be written. Beb incorrectly executes subroutines written in any other programming language. Optimizing routines reduce any source program to exactly three bytes of object code. Conditional compilation causes the computer to compile your program only if it feels like it. Nested infinite loops lock up the CPU, bringing any time-sharing system to its knees. Touted as the first truly portable language, Beb can be installed on any electrical appliance, including such popular machines as VersaTeller and all Toshiba microwave ovens. **Reader Service No. 3.14159265.**

A Trendy Giggle

Here's a new word for you to practice: *gigabyte*. It means "one billion bytes." That is, a thousand megabytes. It's pronounced *with a hard g* followed by a short *i*. I've been reporting disk drives with ever more storage, up to 160 megabytes and counting. We recently received notice from **Capital Systems** that they will be a distributor of **IBIS Systems** IBM-plug compatible 1.25- and 2.52-gigabyte drives for mainframes, primarily for sale to U.S. government agencies. Not for the personal computing crowd yet, nor were prices even quoted, but soon you'll be seeing a lot of that word. And remember the abbreviation *GB*. (Or should it be "Gb," to be consistent with "Mb" for megabyte?) And try to think what you could *do* with all that storage . . . . IBIS also makes a disk drive controller for drives with storage capacities up to *40 Gb*! (And don't ask me if that's *really* a billion, or 1024-cubed . . . .) **Reader Service No. 101.**

A Wonderful Software Marketing Trend

BLOWUP, written in BASIC for the IBM PC, transforms text into inch-high (ten characters by ten lines) block characters on a printer for signs and notices. Not only does the 5¼-inch diskette retail for only $24.95, but it introduces an unusual but welcome distribution philosophy to software marketing that I hope becomes a widespread trend. **Robert A. Murray and Associates** encourage you to copy the software and give it to friends, with the philosophy that "good, inexpensive software will sell itself." They ask only that those who find the software acceptable send them $20, in return for which they will receive a free catalogue of other Murray software products. The honor system: what a refreshing answer to piracy! **Reader Service No. 103.**

Color It Any Color

USI has introduced a 14-inch color monitor with a high resolution, 80-column, 24-line display in seven colors that retails for $399. This is a vast improvement over a TV set, which is not made to be a computer monitor, its low resolution producing blurry characters and fuzzy graphics. USI last year gave this country its first amber screen monitor. Both monitors, and a green phosphor model, are available in 9- or 12-inch sizes. Each USI monitor is housed in a stackable metal case and is style- and color-coordinated with most popular computers, to which they connect with standard phono jacks. **Reader Service No. 105.**

And for a screen dump to printer of your wonderful color displays, the **Transtar 315** prints in color for the same price as many black-and-white dot matrix printers. For $599 you get a four-hammer print head and the ability to print seven colors plus more than 30 shades in a single pass. Color graphics and characters both print at 50 cps. Print speed has little meaning in many printers that intermix colors in multiple passes or multiple strikes at the same spot, but the Transtar 315 does not slow down, pause, or come back. And thus throughput "exceeds machines rated at 200 cps." You can get an interface option right now for Apple][(and for other computers soon) called **PICS** that lets you press a copy button on the printer to get a direct color graphics dump of the screen without using disks, exiting programs, or changing application programs. Transtar also offers a daisy wheel printer, the **Transtar 130**, for $895, that immediately runs from most micros with all word processing packages that have Diablo print routines. You get boldface, underscore, proportional and incremental spacing, parallel or serial interface (one or the other), 300 to 2400 baud, 16 cps throughput, auto paper loading, bi-directional printing, and a six-month warranty. There is also a **Transtar 140**, for which they didn't quote a price, that has all of the preceding, plus 40 cps and a bidirectional tractor feed option. (The printers, by the way, are all made by Seiko's Seikosha Group.) **Reader Service No. 107.**

Adding to Your Color Computer

Enhance BASIC on your TRS-80 Color Computer with **Spectrum Projects' Basic Aid**, a ROM cartridge that sells for $39.95. Automatic line numbering, single-key entry of most commands, redefinition of keys, merging routines stored on tape into a program in memory with automatic renumbering so tape libraries can be built, moving of any program lines anywhere about a program with consequent renumbering of all GOTOs, GOSUBs, and so on, that reference the moved section, are all part of Basic Aid. **Colorcom/E** turns the Color Computer into a smart terminal with on- and off-line scrolling, off-line printing, transfer of cassette files, serial printer support and full or half duplex. The program costs $49.95. They also sell a **Spectrum Stick** joystick that has a swivel ball, long cable, sturdy construction and red LED on-indicator for $39.95 plus $2 S&H. They offer cables that extend the ROMpack port by three feet, add 10

feet to the joystick cable, extend the cassette recorder or printer/modem cable by 10 feet, and connect in line both a joystick and their newest product (so new, they didn't even give us a price, but write them), a light pen. An RS232 expansion cable attaches two devices to the serial port at the same time, permitting a printer and modem to be attached in line, for $19.95. You can use your new equipment to access Spectrum's two bulletin boards, and Colorcom/E to download color graphics: (212) 441-3755 and (212) 441-3766. **Reader Service No. 109.**

The IDE of March

IDE Associates makes a 3.9-inch Winchester disk with fixed or removable cartridges for the IBM PC. Formatted capacity of 5.0 Mb costs $1450 for the internal mount-fixed cartridge version. The removable cartridge versions cost $50 more. The price includes mounting hardware, cables, software and diagnostics diskette, and manual. IDE offers, in ten metropolitan areas, what they claim is "the industry's only free on-site installation service." IDE also sells expansion memory and clock cards for the PC. For all their products, you get a one-year return-to-factory warranty, or, in any of the ten areas, the option to convert the warranty to a fixed-price on-site maintenance agreement. **Reader Service No. 111.**

More Memory at Less Price for Chipmunks

Expand memory on HP's Series 200 (the 9826 and 9836 68000-based computers known affectionately in house as "Chipmunk," and the new 9816 personal "home" computer) with **Eventide's WKBP-4** 256K memory expansion board. At $749, it costs "hundreds" less than HP's equivalent. You can add half a megabyte to the 9816 for under $1500, and over two megabytes to the 9826 and 9836 for, they say, "less than $5200." I'm not sure how they perform this mathematical feat, unless they offer discounts for eight boards . . . . **Reader Service No. 113.**

Altos Talks to Other CP/Ms

InterComm from **Acquis Data** transfers files between an Altos and other CP/M computers. Included is a module to communicate with bulletin boards and public data bases like CompuServe, Dow Jones, and The Source. For $175, you get two diskettes, manual, and communications/null modem cable. InterComm is also available for many other CP/M computers. **Reader Service No. 115.**

If The Code Works, Use It

Version 2.0 of the **Q/C** compiler for CP/M, from **The Code Works**, supports a large subset (all but float and long data types, multidimensional arrays, structures, sizeof, typedef, and casts) of the C language, compatible with Bell Labs' UNIX 7 C, and includes a library of over 50 I/O functions. Q/C produces true native code for Z80 or 8080 for Microsoft's M80 assembler or for Digital Research's ASM or MAC. Assembly language functions can be included in programs. With M80, programs can be compiled as separate modules and Microsoft's L80 linker can be used to combine the relocatable files. Full source code for compiler and library are included, as well as a comprehensive, beautifully typeset 138-page manual. You'll need a 56K CP/M, and either soft-sectored 8-inch single-density or hard-sectored 5¼-inch Micropolis Mod-II format double-density (Vector Graphics MZ) disk. The Q/C disk costs $95. Look for version 3.0 soon, which will support structures, unions, multidimensional arrays, and a few other new features, using the Z80 instruction set, and producing smaller, quicker code. This will be offered as a $12 upgrade to current users. A version for the IBM PC (PC DOS) will also be released at that time. The Code Works also offers what they call the "original" **Small-C** by Ron Cain for $19.95, a much smaller subset of C, with a minimal assembly-language I/O library and an 11-page manual. It needs 48K. Add $3.50 shipping in U.S. and Canada, or $15 elsewhere, and, in California, 6% sales tax. The Code Works charges no license fees for programs created on its compilers and libraries. **Reader Service No. 117.**

Flying a Simulator? *Simulating* Flying?

Now you can do your armchair flying – from the armchair facing your computer, or flying by the seat of your pants – without leaving your terminal. **Flight Simulator** by **Microsoft** for the IBM PC is for novice to experienced pilots. As your skill increases, you add various conditions, such as time of day, cloud cover, season, wind shear, trim, carburetor icing, fuel usage, and even radio navigation. The screen displays an "out-the-window" 3-D perspective view with full instrumentation. You have real-time control of the plane (a single-engine Cessna 182) in several flight conditions, and data that simulate twenty-three airports in four parts of the country (Seattle, LA, Chicago, and New England). You also get **British Ace**, a World War I combat game that test your flying skill and strategy against enemy biplanes defending territory that you must bomb. You'll need a PC with PC-DOS, color/graphics card, 64K, one disk drive, and a monitor (they recommend the standard composite color monitor, but you can use black-and-white or RGB). $49.95. **Reader Service No. 121.**

Legal Copy Service

Port-A-Soft (formerly Disk Copy Service) converts disks between several formats, to or from 5¼- or 8-inch, single-, double-, or quad-density, in formats of various operating systems and many popular computers, for $5 to $15 per diskette, and copies after conversion for $2.50 each. **Reader Service No. 119.**

Other Stuff

Do you own or have access to a DEC system with RT-11, TSX, or TSX-Plus? Do Z80 software development on it with **SATEC Systems' Z80 Cross Assembler**, available for $200 on paper tape, 8-inch floppy, or source listing. **Reader Service No. 123.**

Here's your chance to see if you are smarter than a computer at word games. **Wordtrix** from **Insoft** for the IBM PC has you compete against the machine to find words in a random grid of letters. The computer brings its own dictionary (how can you argue with the guy who brings the bat?) and plays on several levels. $34.95. **Reader Service No. 125.**

Learn about protecting software, copyrighting programs, writing and negotiating your own licensing contracts, limiting liability, and other legal considerations for programmers and software publishers in *Legal Care for Your Software* by Attorney Daniel Remer, published and distributed by **Nolo Press. Reader Service No. 127.**

DDJ

(Continued from page 11)

Since a number of you may be wondering where your package is, we thought that you might be interested in the following letter which we recently received from JRT Systems concerning late deliveries. We see that the letter does not mention adjustments to any of the bugs that have been noted, but we understand that JRT has a new version which they expect to begin shipping in mid-February.

Dear Editor,

We are having delays in the shipping of JRT Pascal to our customers. I have received a number of letters from customers who are very concerned about this. Similar letters have been sent to the editors of the magazines in which JRT Systems advertises.

It has always been our policy to immediately cancel any order on request or make an immediate refund if payment has been processed and shipment not yet made.

As of today we have shipped 10,000 Pascals and have 7,000 orders on backlog. About 6,000 of those are less than six weeks old. A six to eight week shipping delay is not unprecedented in the computer industry, but it is not acceptable to JRT Systems.

How did this backlog develop and what am I doing to correct it? In May of 1982, I cut the price of JRT Pascal from $295 to $29.95. I tested this formula by "mass" mailings of 100 then 400 then 1000 brochures. Sales response was excellent – 10% to 15% – on the early mailings. I was able to very rapidly expand the mass mailings and advertising leading to these approximate sales figures: May - 100, June - 350, July - 700, August - 1400, September - 2000, October - 4000, November - 3000, December - 5000.

In August JRT Systems moved from my home to a small office on Irving Street in San Francisco. By October this office was severely crowded with six people, three phones and two computers. In mid-December we leased 6,000 square feet of space in Mill Valley. The shipping department has now moved from Irving into 1,600 square feet of the new space. In the past two weeks, the shipping staff has grown from one to four full-time people. In this same period, we completed installation of a sophisticated set of new shipping programs which automates, logs and validates every aspect of the shipping operation. Last week we exceeded 500 Pascal shipments per day. With our new system we can exceed 1000 per day.

We still have delays in obtaining copied diskettes rapidly enough, especially in 5¼" formats. Changes planned for the near future will eliminate this problem.

In short, the delay in shipping JRT Pascal is due to the staggering sales growth. We are moving as fast as possible to expand production capacity.

Sincerely,
J. R. Tyson, President
JRT Systems
550 Irving Street
San Francisco, CA 94122
(415) 566-5100

A Few Suggestions

Dear Editor,

First, I want to address the contents of the columns, Of Interest, CP/M Exchange, and 16-Bit Software Toolbox. The first of these columns contains information which I think most *DDJ* readers find redundant. I think that *DDJ* should devote its space to articles and not to condensed press releases which may be found in *BYTE* or any number of other journals. Conversely, I think that the second and third columns have a place in *DDJ*. However, I think that they need some changes. CP/M Exchange needs to deal with topics, not just contain announcements. I would like to see the columnist find topics of interest and discuss them instead of continually referring to RCP/M related announcements. Finally, I think that 16-Bit Software Toolbox should stay away from press releases and concentrate on one (or at most two) topics per month. This would allow the columnist to explore the topics more carefully.

Second, I would like to reiterate my comment concerning the overall quality of the journal. I strongly believe that good material still appears in *DDJ*. However, it is wise to remember that good material appeared in quantity even when the magazine ran with no advertisements. One would expect quality to increase and not for mediocrity to become a serious threat once ads were included.

Finally, I would like to make one additonal comment. I think that articles should be rated by user reaction, and remuneration based thereon. This is already done by *BYTE*.

Sincerely,
Anthony Skjellum
1695 Shennandoah Road
San Marino, CA 91108

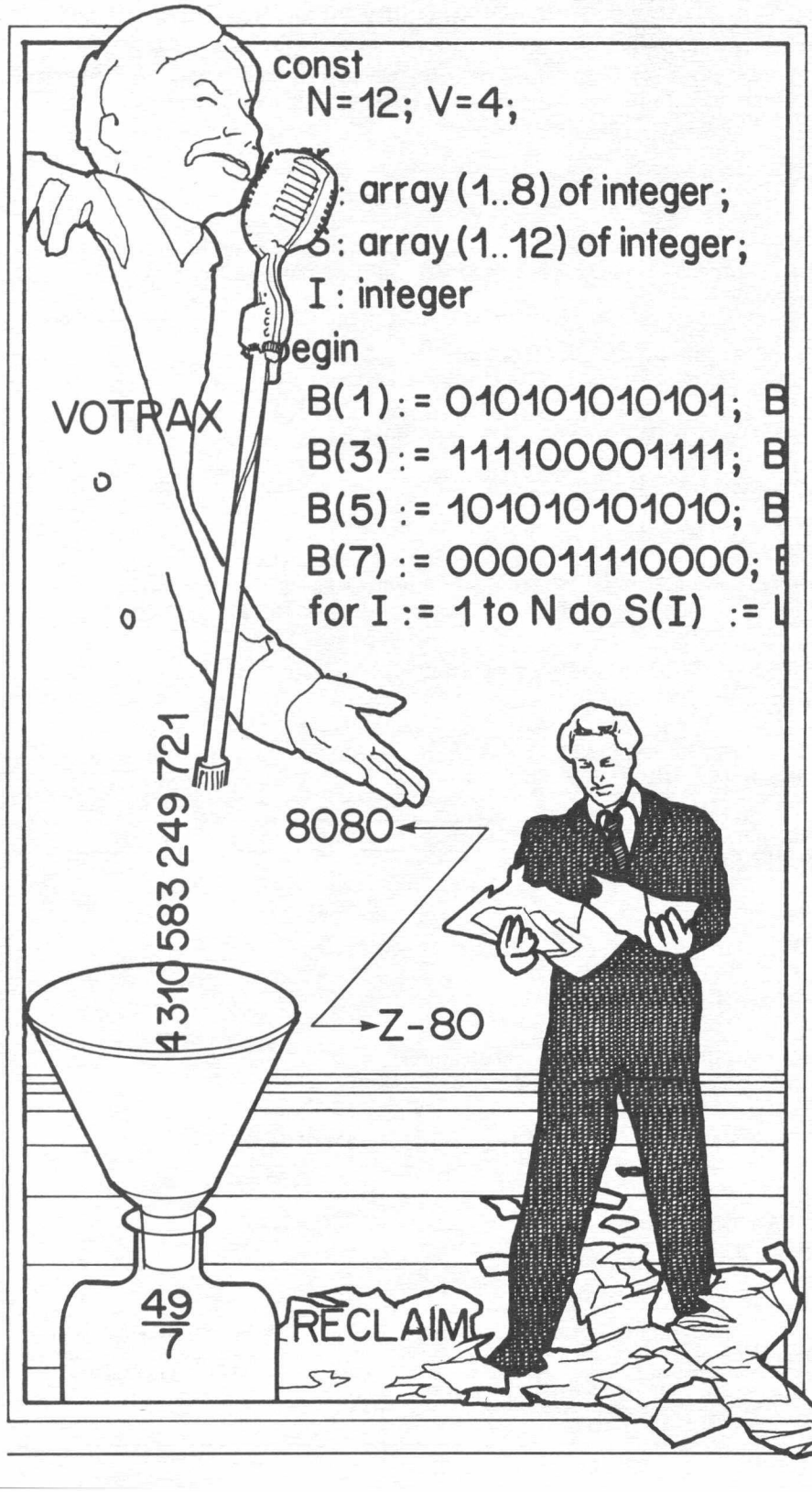
Keeping Us Honest

Joe Williams called in to let us know that we had a minor error in the Small-C compiler listing in the January 1983 issue. The listing portion at the top of page 63 should have preceeded the listing portion on the top of page 62, not followed it. – Ed.

#78 April 1983

Dr. Dobb's Journal

For Users of Small Computer Systems



RECLAIM

Binary Magic Numbers

8080 Fig-Forth Directory & File System

Forth Votrax Driver

. . . and More

DDJ T-Shirt pg. 45

Dr. Dobb's Journal
For Users of Small Computer Systems

April 1983 Volume 8, Issue 4

Publisher/Art Director – Clifford West
Editor – Reynold Wiggins
Editorial Consultant – Marlin Ouverson
Contributing Editors –
Dave Caulkins, Dave Cortesi,
Ray Duncan, Gene Head,
Michael Wiesenberg
Marketing Director– Craig Harper
Marketing Manager – Beatrice Blatteis
Advertising Director – Carl Landau
Advertising Sales – Doug Millison
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Terri Marty, Linda Marohn
Production Manager – Barbara Ruzgerian
Production Assistant – Jane A. McKean
Typesetter – Paula Fairchild
Cover Illustration – Al McCahon

© 1983 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: DeWitt S. Brown, Burks A. Smith, Robert M. Connors, Robert C. Luckey, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Frank Lawyer, Rodney Black, John Brodie, Thomas Davis, Jan Steinman, Ronald E. Johnson, G. Hunter, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III, Infoworld, Stan Veit, Western Material Control, S.P. Kennedy, Ed Moran. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

CONTENTS

ARTICLES

14 RECLAIM – A File Reclamation Utility for Destroyed Directories
by Walter V. Murphy
Ever started formatting a disk, only to remember that it contained important files? Or had a runaway program run over your directory? All is not lost! With RECLAIM you can get those files back.

24 Binary Magic Numbers
by Edwin E. Freed
There are certain integers for each number base that have special or unusual properties. Author Freed discusses how binary "magic numbers" can be very helpful to programmers.

38 An 8080 Fig-Forth Directory and File System
by Alfred J. Monroe
This Forth program overlays a directory-and-file system onto Fig-Forth for North Star DOS. Forth code makes the program highly portable.

55 SAY" Forth Votrax Driver"
by C. Kevin McCabe
Designing software to interface your system to such things as a Votrax speech synthesizer can be difficult with conventional languages such as BASIC. Forth's modularity and expandability prove useful in the interface presented here.

60 A TRS-80 Z80 to 8080 Translator
by Anthony T. Scarpelli
Working from Robert W. Dea's original translator (published in *DDJ* No. 45), author Scarpelli has produced a Microsoft BASIC version that is faster and comes with a flowchart.

DEPARTMENTS

6 Letters
6 Editorial
9 CP/M Exchange
Part Two of CP/M disk I/O
76 Dr. Dobb's Clinic
Disk Drives Us Crazy, Counts and Measures,
Working on CP/M Plus with RMAC and LINK 80
79 16-Bit Software Toolbox
Reflections on CP/M 83, Intel 8087 News, Preview of CP/M-68K,
Improved Square Root Routines
83 Software Review
Unica & XM-80
86 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

LETTERS

Xanadu Re-Edited

Dear Dr. Dobb's:

Thank you very much for publishing our piece on the Xanadu Hypertext System in the January issue of *Dr. Dobb's*. While we were delighted to get word of our system in front of an audience, the editorial process somehow managed to introduce a couple of errors which deserve correction.

First of all, the article says that the Xanadu System is a currently available product. This is not entirely correct, and that was not how we wrote it. We *are* now marketing the system for large scale commercial applications, but we are doing so solely as a part of custom systems being produced on spec. While the system *does* work, it is as yet under development, and several more months of effort will be required before it will be a marketable product available "off the shelf." We are still working on various more advanced portions of the system, including the versioning facility and historical trace. We have been giving demonstrations regularly and are now looking for our first major customer. We are prepared to offer custom systems for applications such as engineering project management, software maintenance and source code management, and business document handling. We are currently quoting delivery times of six months to a year for such systems. Anyone interested should contact us for details, quotes or demonstrations.

Second, the article is credited to me. While I was the person who sent you the Xanadu material, the piece that you selected for publication, out of the two-hundred or so pages that we sent, was written by our Executive Vice President, Chip Morningstar. Since it was he who put in the hours writing it, he should get the credit for it.

We would like to talk with anyone interested in developing frontend systems. A preliminary protocol for frontend/backend interaction is available for the asking to anyone seriously contemplating frontend work. Our frontend development currently takes place on our SUN (trademark SUN Microsystems Inc.) workstation in C under UNIX (trademark Bell Laboratories). A copy of the current frontend is also available to anyone seriously interested. Most of our energies have been devoted to getting the backend working, and there are a lot of interesting human-machine interface problems yet to be attacked. Again, interested parties should contact XOC for details.

Thank you again.

Sincerely,
Roger Gregory, President
XOC
P.O. Box 7615
Ann Arbor, MI 48107

OK to Repeat

Dear Doc,

I'm writing this on the last day of 1982. It has been a couple of years since I last wrote, but a passing comment in J. H. Peters' letter in the January 1983 issue has moved me to the tyepwriter.

Mr. Peters' letter was specifically concerned with JRT Pascal but, as an aside, included the comment that "Pascal includes what is probably the worst programming practice known to modern science: running once through a loop before making any tests (REPEAT. . . UNTIL)." This comment pushed one of my hot buttons. That alone would not have sparked a letter except that this particular hot button has happened to be exercised repeatedly in the last few months.

REPEAT. . .UNTIL is not a programming practice at all. It is a tool provided by the language designer to be used, or not used, as may be deemed appropriate by programmers working in the language. This tool, like any other, may be left to rust in the toolbox or it may be used as needed when its use is deemed appropriate.

One good example of an appropriate use of this tool is provided by Kernighan and Ritchie in *The C Programming Language*. In their itoa(n,s) function, it is desired to convert an integer, n, to an ASCII

EDITORIAL

Welcome to our second ninety-six page issue! We are pleased to bring you more editorial pages than ever, and expect even more in the second half of 1983. As we grow, you will find more of the high-quality material that you have come to expect from *DDJ*. The increase in size will help ensure that the reader forum will continue to be provided in our pages. This is one of the things that separates us from the crowd – the difference between editors and experts.

* * *

This issue's CP/M Exchange is the second of a two-part series on CP/M disk I/O. It marks Gene Head's last month as columnist for the Exchange, though he will continue to operate his RCP/M system. We would like to thank Gene for all his fine contributions over the past months and look forward to continued input from his Oregon Head Quarters. Gene is passing the pen to Bob Blum. No stranger to *DDJ* or to the Exchange, Bob provided the current series on disk I/O, and he will begin a series on CP/M Plus in the May issue. We are glad to welcome him aboard.

* * *

A couple of months ago we mentioned the possibility of condensing some listings in order to highlight exemplary or instructive sections of code. The motivation was to get *more* information to you, not less. We fear this may have given rise to the misinterpretation that we were making a significant shift toward eliminating listings. Not so! *Dr. Dobb's* has always been a forum for software and programming ideas. In keeping with our tradition, we will, of course, continue publishing complete listings. . . . Are those sighs of relief we hear?

Reynold Wiggins
Editor

string at s. Leading zeroes are to be suppressed *unless* n has a value of zero, in which case a single ASCII zero should be generated. This exceptional case is neatly accommodated by passing once through the loop before testing for a zero value in n.

Best regards, and best wishes in the new year.

William T. Mitchell
P.O. Box 94978
Schaumburg, IL 60194

Hardware Help

Gentlemen,

I would like to see a hardware article on making a dynamic RAM board using an Intel 8203 dynamic RAM controller (for 64K DRAMs) or 8207 (for 256K DRAMs). Intel has application notes on just such topics, but they provide only block diagrams (all of the ICs are listed) and I'd like to see a complete schematic of a working board.

I've been looking for information on connecting a Model 33 teletype to my TRS-80 Color Computer and my Sinclair ZX81. I've been able to find a general RS232 to 60mA current loop interface diagram, but would like more specific information (my Model 33 TTY is set for 20mA current loop). I'm sure many people have used Model 33 TTYs as printers, but I've been unable to locate any (sorry this note isn't printed!). I'd greatly appreciate any help you could give me on this matter (both hardware and software).

Yours truly,
Timothy J. McIlwee
Rural Route 2, Box 462A
Dundee, IL 60118

Forth Loader Fix

Dear *Dr. Dobb's*:

I encountered some problems using the Forth Relocating Loader which was published in your September 1982 issue (No. 71). Upon examination, I found that the program was doing signed comparisons of addresses instead of unsigned comparisons (i.e., using < and > instead of U< and U>). I corrected this in screens 10 and 11, and in 14 through 19, and everything worked fine after that.

Thanks, and keep up the good work on your magazine.

David J. Sieber
Star Computer Systems
20600 Gramercy Place
Torrance, CA 90501

DIR.ASM Problems?

Dear *DDJ*:

DIR.ASM, published in the CP/M Exchange column in February 1983, probably should not have been published at all; but since it was, it should have had the warning: "Use this program at your own risk. It is untested."

I wrote the program in response to a request to give the readers code. After hurriedly submitting the untested program, I found it to be machine dependent in some cases and told the editor about the problems. I was assured a disclaimer would be included, and was told that the code was a good starting point for a better directory utility and maybe a challenge to hackers to get operating on specific systems.

Since, then, I found SD-41.ASM in the public domain. This is a far better directory utility than DIR.ASM with more features and faster execution not to mention fewer (zero) known bugs.

To those trying to get DIR.ASM working, I suggest you scrap it and get a copy of SD-41.ASM (and SD-41.DOC) from any RCP/M system. I apologize for any inconvenience and in the future will not make the mistake of releasing code without complete field testing.

Sincerely,
Gene Head
Head Quarters
2860 NW Skyline Drive
Corvallis, OR 97330

Hi-Res Vector Display Revisited

Dear *DDJ*:

Following my letter in the September 1982 issue (*DDJ* No. 71), I have received many letters concerning the color graphics display I described therein. I cannot spare the time to give individual replies but do intend to write up the system for publication when it is finally perfected. There is still a lot of work to do before it can be confidently offered to others to duplicate, mainly in cutting the enormous cost – e.g., the special CRT used costs over $1000 on a one-off basis.

Have other 68000 users any ideas on standardization of memory maps, disk formats, etc. to allow software interchange? There seem to be no standards at all at the moment and I fear we will have a similar situation to that which now applies to the 6502, with many machines having no interchangeability of software whatever. My own choice, for what it's worth, is to place ROM between 000000 and 00FFFF, I/O ports from 010000 to 01FFFF and RAM in the rest of the address space. I use the IEEE-696 S-100 bus, which is very suitable for this system and gives good hardware interchangeability – I have processor boards with other CPUs which I can use when I need to.

Finally, does anyone have any information concerning a Wangco 8201 floppy disk controller board and an MFE model 250B digital cassette deck? Wangco no longer make the former and cannot supply any information and I've no idea who MFE are at all. Any info would be very welcome.

Sincerely,
Greg Trice
1131 Sandhurst Circle, #111
Scarborough, Ontario
Canada M1V1V5

Ruzinsky Re-Corrected

Dear Readers,

Steven Ruzinsky wrote in to tell us that a production error had caused an incorrect correction to his equations on page 10, column 3 of the March issue. Lest any be confused by the resulting "non-equation," the text *should* have read:

```
Let U = Angle
S1 = .9999999999999999
V1 = S1*[FABS(FABS[FPREM
       (U,2*PI)] - PI) - PI/2]
COS(U) = Y(V)/FSQRT( X(V)*X(V)
         + Y(V)*Y(V) )
```

Our apologies to Mr. Ruzinsky and anyone who was inconvenienced.

The Editor

DDJ

Circle no. 78 on reader service card.

CP/M EXCHANGE

by Gene Head

Bob Blum will take over this column beginning next month. Bob has brought considerable technical expertise with this two-part series on the CP/M disk and plans to begin a series on CP/M+ next month.

Take the time to read and re-read this month's column. It's all meat and no fat but you may need a pencil and paper to do some quick calculations. It was a real eye-opener for me. Bob knows his stuff!

Last month we reviewed how CP/M allocates and maintains disk space. Our discussion this month will continue by exploring first the file control block (FCB) and then the various BIOS tables used to describe the disk system and their interaction with CP/M.

When accessing a disk file through BDOS, a properly initialized FCB must be used. The open routines use the file name and type fields as the search argument for the desired file. Upon a successful match, the allocation information from the stored extent is copied into the FCB for future file operations. During input/output (I/O), various fields of the FCB are used to maintain an updated copy of the file's status.

When writing to a file, if the FCB's addressing range is exceeded or the file is closed, the current FCB is written to the disk directory after the last four bytes are discarded. This truncation maintains compatibility with earlier releases of CP/M and also eliminates unneeded information. It should be noted that the names, FCB and extent, can be used interchangeably, although FCB typically refers to the memory resident file control block, while extents are disk resident.

Referring to Figure 1, the first byte of the FCB, DR, has a dual purpose depending on whether the FCB is memory or disk resident. When the FCB is in memory, the DR entry (when non-zero) designates which disk drive is to be auto-selected. After the FCB is written to disk, the DR entry reflects the file user number. The next 11 bytes contain eight bytes for the file name and three bytes for the file type. The high-order bits of the first two bytes of the file type are used to indicate if the file is read-only or a system file. Only two of the next three bytes are important to this discussion. The first byte, EX, contains the current extent number in the range of 0 to 31. If the file expands beyond 32 extents, the EX byte is zeroed and the third byte, EC, is incremented. These two bytes can be visualized as containing a nine-bit number with the EX byte containing the low-order five bits and the high-order four bits present in the EC byte. To verify the maximum file size of 8,388,608, multiply each extent of 16,384 bytes by 512 extents.

Another way to view the association between these two fields is that the EX byte contains the extent number within a group contained in the EC byte. Continuing along to the remaining fields, as each 128-byte sector is written, the record count byte, RC, is incremented by one until 128 is reached. When this occurs, the FCB is written into the directory and a new FCB is automatically prepared for additional file operations. We can now verify that each extent can address 16,384 bytes by multiplying 128 sectors by 128 bytes per sector. The allocation map area of the FCB, AM, provides room for 16 bytes of data allocation group numbers. Each allocation group number can be either one or two bytes in length depending on how many allocation groups are contained on the disk. Further attention will be given to this later in the discussion on disk parameter blocks. The remaining four bytes of the FCB are used for sequential and random access into a file and are left for independent study.

As delivered by Digital Research, CP/M is configured for the MDS-800, which is not in common use. For *that* reason we will assume that CP/M is unconfigured as distributed. Before studying how to configure CP/M for a particular disk system it may be helpful to examine an overview of all the integral parts of the BIOS. Figure 4 (page 12) shows us that the CBIOS begins with a series of sixteen jumps followed by one disk parameter header (DPH) for each disk drive attached to the system. The DPH contains address pointers to various other tables that completely describe the disk formats that can be used on this particular system.

From Figure 2 (page 10), the first field in the DPH, XLT, is a two-byte pointer to

FILE CONTROL BLOCK

OFFSET	SYMBOL	BYTES	DESCRIPTION^S
0	DR	1	Drive number: 0 = Default drive 1 = Auto select drive A ----- ----- 16 = Auto select drive P
1	FN	8	File name in upper case ASCII. Padded on right with blanks if necessary.
9	FT	3	File type in upper case ASCII. High order bits contain additional information. BYTE 9 = 1 Read only file BYTE 10 = 1 System file. File will not be listed during a directory list.
12	EX	1	Number of current extent in use.
13	WA	1	Workarea.
14	EC	1	When extent number, EX, exceeds 32 this byte acts as an overflow area.
15	RC	1	Record count. Number of 128 byte records contained in this extent.
16	AM	16	Allocation map. Allocation group number storage area.
32	CR	1	Current record. Number of record currently be accessed.
33	RR	3	Random record. Record number to be used for accessing the file randomly.

Figure 1.

the physical sector translation table. Through this table it's possible to translate a logical sector to its physical equivalent. Many of today's systems allow different disk formats to be used interchangeably on the same drive. For this reason, when logging in a disk, the BIOS will probably access a known part of the disk to determine the format and place the address of the corresponding disk parameter block into DPB and the appropriate translation table address into XLT.

The next two bytes are used for a workarea by BDOS followed by a two-byte pointer, DIRBUF, to an area of memory 128 bytes long which is used for directory operations. All DPHs can point to the same DIRBUF because only one directory operation will be active at a time. For example, when a file is opened, the directory is read into the buffer pointed to by DIRBUF one sector at a time. If the desired file is found, an index number is returned to the requester so that additional information can be extracted. CSV points to a workarea used for calculation of the directory checksum. This checksum is used to detect when a disk has been changed without a warm start. Each disk drive must have its own CSV workarea. The DPH ends with a two-byte pointer, ALV, to a savearea which will contain the allocation map after a disk is logged in. A separate ALV savearea must be reserved for each disk drive on the system.

The disk parameter block (see Figure 3) contains all the information necessary to describe the disk format in use. The first entry, SPT, contains the total number of logical sectors per track. This number may not equal the number of physical sectors actually written on each track. For example, TARBELL double-density format uses sixteen 512-byte sectors per track. Since there are four 128-byte sectors per physical sector, SPT would contain 64.

Before discussing the next three fields, a few decisions must be made. First, decide how large the data allocation block should be. On systems where the total capacity of any one disk is less than 225K, block sizes from 1K to 16K can be used. On larger capacity formats you are limited to selecting block sizes 2K and larger. Tradeoffs between block sizes are subtle. If you consistently create small files, a smaller block size will probably be best suited to your needs. Larger block sizes have the advantage of keeping more sequential data available without excessive head movement.

Next determine how many data blocks can be allocated. This is a function of the total capacity of the disk format being used minus the number of reserved system tracks divided by the data block size. This value is placed in DSM.

With these decisions made, the tables in Figure 5 (page 12) are used to provide values for BSH, BLM, and EXM. The extent mask value further redefines how much storage can actually be addressed by each FCB. For example, disks with less than 255 data allocation groups only require one byte for each group pointer. If the allocation groups are 2K in size then only eight bytes are used for each 16K addressed leaving eight bytes of allocation map unused in the FCB. By coding EXM with 1, two logical extents can be stored in each physical FCB extending its addressing range to 32K.

The situation becomes more confused when there are more than 255 allocation groups. In this case each pointer requires two bytes. In any case the tables in Figure 5 will provide the proper entries to be used. The total number of directory entries, DRM, is also a function of the data block size. Each directory entry is 32 bytes long. If the data block size is 1K then 32 directory entries will be contained in each block.

Typically two data blocks are reserved for directory use. DRM would then be

DISK PARAMETER HEADER

OFFSET	SYMBOL	BYTES	DESCRIPTION^S
0	XLT	2	Address of the logical to physical sector translation table.
2		6	BDOS workarea.
8	DIRBUF	2	Address of a one sector workarea used for directory operations.
10	DPB	2	Address of the disk parameter block for this drive.
12	CSV	2	Address of a workarea used to calculate the directory checksum to detect changed disks.
14	ALV	2	Address of a savearea used for this drives allocation map.

Figure 2.

DISK PARAMETER BLOCK

OFFSET	SYMBOL	BYTES	DESCRIPTION^S
0	SPT	2	Total number of 128 byte sectors per track.
2	BSH	1	Data allocation block shift factor as determined by the data block allocation size.
3	BLM	1	Data allocation block mask.
4	EXM	1	Extent mask determined by the number of data blocks allocated and the data allocation block size.
5	DSM	2	Total number of data blocks allocated to this drive.
7	DRM	2	Total number of directory entries allocated for this drive minus 1.
9	ALO AL1	1 1	Table of 16 bits set on from left to right indicating allocation groups reserved for directory usage.
11	CKS	2	Size of the directory check vector.
13	OFF	2	Track offset to logical track zero.

Figure 3.

coded with 63. The decision on how many data blocks to reserve for the directory is reflected in entries AL0 and AL1. Given that two data blocks are reserved for directory use, the two high-order bits of the 16-bit quantity formed by combining AL0 and AL1 would be set on. This value is used when a new disk is logged in to mask the allocation vector table to prevent any damage to the directory data blocks. The next entry, CKS, should contain the total number of directory entries in use. Each time a disk is selected after it has been logged in, the directory is checked against the stored checksum to determine if the disk has been changed without a warm start. If a hard disk is in use it is not necessary to provide for directory checking.

By using the information contained in the disk parameter block, you can move about the disk system at will. To calculate the starting track and sector numbers from a known allocation group number requires only a few steps. First multiply the group number by BLM plus one. This can easily be done by shifting the group number left the number of times contained in BSH. This results in supplying the actual sector number as if all the sectors were sequentially numbered. Now divide the result by SPT and add OFF to the quotient. The remainder is the starting sector number while the quotient is the starting track number.

To calculate a group number with known track and sector numbers does not require any more effort. Subtract OFF from track then multiply the track number by SPT and add the sector number to the quotient. Now divide the quotient by BLM plus one.

In a very short time we have covered the same material that has been presented in a number of books. I suggest that you use the information presented here while reviewing a BIOS listing. Next month the features of CP/M Plus will be compared to those provided by CP/M 2.x.

DDJ

```
                BIOS CONSTRUCTION

JMP         COLDSTART
JMP         WARMSTART
        ------
        ------
JMP         LISTSTAT
DB          (Disk Parameter Header for Drive 1)
DB          (Disk Parameter Header for Drive N)
            X     X     X     X     X
            X     X     X     X     X
 XXXXXXXXXXX      X     X     X     X
 X            XXXX     X     X     X
XLT             X       DPB   CSV   ALV
             DIRBUF
```

Figure 4.

```
DETERMINING  BSH - BLM UNITS

          BLS        BSH        BLM
         1024          3          7
         2048          4         15
         4096          5         31
         8192          6         63
        16384          7        127

                BLS        EXM
               1024          0
               2048          1
               4096          3
               8192          7
              16384         15

If DSM > 255    BLS        EXM
               1024        N/A
               2048          0
               4096          1
               8192          3
              16384          7
```

Figure 5.

RECLAIM

A File Reclamation Utility for Destroyed Directories

Editor's note: The author informed us that he wrote this utility in self-defense, after an errant program had destroyed his directory of a megabyte of important files. It took him about fifty man-hours, including the writing of the program, to completely recover all those files. The source code for this utility is written for use with Ron Cain's version 1 of the Small-C compiler. It has not yet been tried with J. E. Hendrix's upgrade, Small-C version 2 (see DDJ *Nos. 74 and 75), but it should be upwardly compatible.*

This utility is available from the author on 8-inch, SSSD, CP/M-format diskettes for $30, which include the source code, additional documentation on use and configuration, and an executable version (for CP/M 2.2). For other formats, send inquiries to the author.

Many people who make frequent use of microcomputers have had the disheartening experience of loss of data in a disk system. This can be due to operator error, program error, or hardware malfunction. Experienced people usually will not offer any sympathy to those who lose important data because it should have been backed up on some removable media. Despite this common-sense rule, people still fall prey to various time and productivity pressures and occasionally lose data without having made a backup. In some cases the data lost can be extensive and valuable and warrant an effort in recovery. This is why disk patching programs exist; many can be found in the commercial marketplace and the public domain. Mostly they are useful if files have been inadvertently erased, or a sector develops an error and becomes unreadable due to a variety of possible reasons. Erased files can be reclaimed if the disk is not subsequently written to because erasing merely puts a flag in the directory entry for that file which says the file is erased – the data is not literally erased. Therefore that flag can be changed and the file "un-erased."

by Walter V. Murphy

Walter V. Murphy, Compucations, 212 Northwood Avenue, So. San Francisco, CA 94080.

Copyright © 1983 by Walter V. Murphy. RECLAIM is intended for non-commercial use only. Any commercial use, without the author's written permission, is prohibited.

The problem is on a more serious level, however, if the directory is more extensively damaged. In a CP/M-based system, storage space on a disk is allocated in blocks. A block is always a multiple of a sector, typically 8, 16, or 32 corresponding to 1K, 2K, or 4K byte units of allocation where a sector contains 128 bytes. These blocks are given sequential reference numbers and these numbers are placed in a list in the directory and associated with a given file name. The blocks assigned to a file need not be contiguous and it is permitted for them to be scattered in any order on the disk. These block numbers are the single most important piece of information the operating system deals with in maintaining a file. Clearly if the list of block numbers is destroyed for an individual file, the problem of reclamation is worsened, but if the directory for an entire disk is totally destroyed, the magnitude of the problem can be overwhelming, particularly if the disk was large and almost full.

The program presented here deals with this kind of problem. The reclamation process cannot be fully automated; a qualified person familiar with the types of data on the disk must still inspect the individual blocks and determine which to save and the order in which to group them. This inspection process requires one of the sector examination programs. One which I have used is DISKDOC, written in C and published in *DDJ* No. 66. It would make a nice companion to RECLAIM. The next step is to retrieve these multiple-sector blocks from the disk and write them into a file on another disk. RECLAIM is almost indispensible at this step in all but the most trivial cases.

How RECLAIM Works

RECLAIM is written in Small-C v.1 and intended for use on CP/M-compatible operating systems. It is run while logged into the disk which is to be reconstructed and it writes out files containing a single block each onto a different disk. RECLAIM, as it is presented here, can only be run on systems compatible with CP/M 2.0 or better. This is because RECLAIM gets specific file system information from the disk parameter block. However, it could be modified to get the file system information directly from a CP/M 1.4 system. RECLAIM makes direct BIOS calls in order to fill a block-sized buffer with all the sectors of the requested block and then uses the file system write command to create the file. When RECLAIM is run, it expects input from STDIN, the standard input device, which can be a file using Small-C's directed console input. This input is one text line for each block to be extracted from the disk. Each line contains the track number, the starting sector number of the block, and the filename including drive specifier in which to write the block. Whenever a large quantity of blocks is to be reclaimed, the list of blocks should be put into a text file and presented to RECLAIM through console redirection. The RECLAIM program and the block list file should never be written to the disk which is to be reclaimed; instead they should be accessed from another disk by using drive specifiers. The following example should be very clarifying.

The block list file "INPUTFIL" is a text file on A: containing:

```
2,17,a:file.01
2,33,a:file.02
2,49,a:file.03
5,33,a:file.04
23,47,a:file.05
54,1,a:file.06
```

The damaged disk is in drive B:. RECLAIM is on drive A:. Drive B: should be logged in as shown in Figure 1 (page 16). If at any point in the process an error occurs, the program trys to give appropriate error messages and then stops.

The overall recovery process can be summarized as follows:

(1) Use a disk sector examination program such as DISKDOC to inspect individual sectors for data characteristics. For example, it should be very easy to differentiate .COM files and text files by inspection as most disk sector examination programs also show the ASCII test equivalent of the bytes whenever possible.

(2) Decide which blocks will be kept and which are unimportant. Make a list of blocks by track and starting sector number. It is not necessary to include every sector you wish to save, only the starting sector of every block. The entire block will be automatically retrieved by RECLAIM. The block structure starts on your disk at the next sector immediately after the directory. In a typical double-density system with 64 directory entries, the first block is on track 2, sector 17 and blocks are 16 sectors long. At this time make any notes you can identifying the contents of the block so it will be easier

later to combine the blocks in a way that makes sense.

(3) Using a text editor, create a file which contains the list of blocks you wish to reclaim.

(4) Run the RECLAIM program while directing the input text file into it.

(5) Examine any of the files created which are text and concatenate them appropriately while giving them meaningful names. If some files are purely data then you will need information on the structure of that data in order to piece it back together. In certain databases it is not unusual to find a field which represents the date, a stock number, or line item number. Usually there is either a fixed record length for the database or a consistent identifying pattern at the end of each record such as a carriage-return line-feed pair. Knowing this and using a debugger such as DDT, you can decide which blocks are associated with each other. Later, blocks can be concatenated – using your text editor if they are text, or using PIP if they are any other kind of data. Text data can be verified by printing it out and looking at its appearance. Source code to programs can be verified by assembling or compiling them and database files can be verified by using them in a test application.

I think it is good advice to not attempt to erase or in any way modify the disk being reconstructed until you have not only retrieved all the blocks of interest but are also confident that the order in which you have restored them is correct. Good luck if you attempt a large recovery procedure of this sort. Although it is tedious and time consuming, you have nothing to lose. The best advice is, as always, keep backups of all important data and avoid ever having to reconstruct it.

DDJ

(Listing begins on page 18)

```
B>
B> a:reclaim <a:inputfil
   COMPUCATIONS Disk block RECLAIMation utility, 12Aug82.
   This disk has 54 sectors per track and 16 sectors per block.
   Track: 2, Starting Sector: 17, Filename: a:file.01
   Track: 2, Starting Sector: 33, Filename: a:file.02
   Track: 2, Starting Sector: 49, Filename: a:file.03
   Track: 5, Starting Sector: 33, Filename: a:file.04
   Track: 23, Starting Sector: 47, Filename: a:file.05
   Track: 54, Starting Sector: 1, Filename: a:file.06
   Successful completion. Goodbye from RECLAIM.
B>
```

Figure 1.
RECLAIM Log On for Drive B:

RECLAIM (Text begins on page 14)

```
/*      RECLAIM.C

        A utility for crashed disk data reclaimation.

        By Walter V. Murphy   (415) 952-4484
        Compucations Computer Engineering
        212 Northwood Avenue
        South San Francisco, Ca. 94080

        RECLAIM is intended for non-commercial use and
        distribution.

This program accepts text input lines from STDIN containing
track, starting sector and filename information. The indicated
disk blocks are read and assembled into files on another disk
using the specified names. Many lines of input can be given to
the program by using input redirection.

The format of the input is as follows:
                Track,Sector,Filename.ext

Example:        64,16,PROGRAM.ASM
                65,48,DATAFILE.A01

When using directed input the command line looks like this:
                RECLAIM <INPUTFIL

Where INPUTFIL is a text file containing the input lines.
This allows generating a long sequence of disk blocks to
be read using a text editor.
*/

#define ERROR   -1      /* These declarations would usually be  */
#define TRUE    1       /* in a file called STDIO.H             */
#define FALSE   0
#define NULL    0
#define HOME    30
#define ESC     27
#define CR      13
#define LF      10
#define SECLEN  128     /* sector length */

/*
The external declaration is here because the Small-C this program
was intended for generates M80 assembly language source code.
*/
extern  stdin,stdout;

char *bufr;             /* block buffer */
int track,              /* for bios call */
    sector,             /* for bios call */
    blksiz,             /* number of sectors in a block */
    spt;                /* number of sectors per track  */
```

```
char filename[100],        /* holds the filename, s is the input line, */
     s[100];               /* These don't need to be so long but they  */
                           /* are just for protection sake. */


main()
{    char *x,              /* x is a working input line pointer. */
          *addr;           /* addr is used for giving the DMA to cp/m. */
     int i,                /* i is for general use. */
         fd;               /* fd is a file descriptor number. */

     i=bdos(12,0);                    /* get cp/m version number */
     if (i/0x100) { signon();
         printf("\n  This program will not work with MP/M.",1); exit(1); }
     if ((i%0x100) < 0x20) { signon();
         printf("\n  This program will not work with less than CP/M 2.0.",
             1);
         exit(1); }

     if (!sysparms()) {signon(); /* get file system parameters */
                 printf("\n  Unknown system error.",1); exit(1); }

                                  /* if memory is available- allocate it */
     if ((bufr= malloc(SECLEN*blksiz))==NULL) { signon();
                 printf("\n  Not enough memory for buffer.",1); exit(1); }
     clrscreen();
     signon();
     printf("\n  This disk has %d sectors per track ",spt,2);
                 printf("and %d sectors per block.\n",blksiz,2);

     gets(s);                     /* get line from stdin */
     while  (*s!=NULL) {
         x=s;
         track=atoi(x);           /* first number in line is track */
         while (isdigit(*x)==TRUE) ++x;
         while (isdigit(*x)==FALSE) ++x;
         sector=atoi(x);          /* second number in line is sector */
         while (isdigit(*x)==TRUE) ++x;
         while (isalpha(*x)==FALSE) ++x;
         strcpy(filename,x);      /* the filename is last */


/* If you want the second field in your input line to be a track
   relative block number rather than sector number then include
   the following line:
         sector= ((sector-1)* blksiz) +1;
*/
         printf("  Track: %d, Starting Sector: %d, Filename: %s\n",
                 track,sector,filename,4);
         if (sector > spt) {
                 printf("  The sector requested is too big.",1);
                 exit(1);
                 }
```

(Continued on next page)

```
        gets(s);                    /* get line from stdin */
        i=0;  addr= bufr;           /* now fill the block buffer */
        while ((dread(track,sector,addr) != ERROR) && (i < blksiz)) {
                addr+=SECLEN; i++;
                if ((sector+=1)>spt) {sector=1; track+=1;}
                }

                        /* blksiz is number of sectors to write */
        if ((fd=creat(filename,0))==ERROR)
           {printf("  Unable to create that file for you.",1); exit(1);}
           else i=write(fd,bufr,blksiz);

        if ((i==ERROR) || (i!=blksiz))
                {printf("  Error while writing that file.",1); exit(1);}
        close(fd);
    }
    printf("\7  Successful completion. Goodbye from RECLAIM.",1);
}
                /* End of main procedure */


signon()
{
printf("  COMPUCATIONS Disk block RECLAIMation utility, 12Aug82.",1);
}

/* Procedure to read a sector using the BIOS. */
/* Returns true or false for success or failure. */

dread(trk,sec,addr)
    int trk,sec;
    char *addr;
{   bios(10,trk);               /* select the track */
    bios(11,sec);               /* select the sector */
    bios(12,addr);              /* give DMA address */
    if (bios(13,1)) return 0;   /* perform the read */
    return 1;                   }


/* Procedure to write a sector using the BIOS. */
/* Returns true or false for success or failure. */

dwrite(trk,sec,addr)
    int trk,sec;
    char *addr;
{   bios(10,trk);               /* select the track */
    bios(11,sec);               /* select the sector */
    bios(12,addr);              /* give DMA address */
    if (bios(14,1)) return 0;   /* perform the write */
    return 1;                   }


/* get system parameters from the BIOS. Set values for
   sector count and number of sectors in an allocation block.
   return true if ok.  */
```

```
sysparms()
{    int *dph,i;
     char *dpb,*bsh;
     i=bdos(25,0);
     if ((dph=bios(9,i))==0) return FALSE;
     dpb=dph[5];                    /* get disk parameter block */
     spt=(dpb[1]*0x100)+dpb[0];     /* get # sectors per track  */
     bsh=dpb[2];                    /* get block shift factor   */
     blksiz=1;                      /* 2**bsh is # sectors per block */
     while (bsh) {blksiz=blksiz*2; bsh--;}
     return TRUE;
}

/* carriage return, line feed
*/
nl()
{    bdos(2,CR); bdos(2,LF);
}

/* clear screen
*/
clrscreen()
{    bdos(2,HOME); bdos(2,ESC); bdos(2,'Y');
}

/* Interface to call the bios.
*/
bios(fun,arg)
     int fun,arg;
{    char *ofs;
     ofs=(fun-1)*3;
#asm
          pop       d         ;offset
          pop       h         ;ret
          pop       b         ;arg
          push      b
          push      h
          push      d
          lhld      1         ;pointer to bios
          dad       d         ;add offset
          call      gohl      ;call indirect thru HL.
          xchg
          mov       l,a
          mvi       h,0
          pop       b         ;ofs
          push      b
          mov       a,c
          cpi       (9-1)*3 ;select disk function ?
          jnz       biosout
          xchg                ;if yes put the dph addr in DE.
biosout:
#endasm
}

#asm
gohl:    pchl
#endasm
```

End Listing

Binary Magic Numbers

Some Applications and Algorithms

by Edwin E. Freed

Edwin E. Freed, Mathematics Department, Harvey Mudd College, Claremont, California.

This article presents a set of constants that enhances many different kinds of binary calculations. These "magic numbers" can be used to determine the positions of bits within words; to reverse, permute, and map bits within words; to compute sideways sums and parity; and to convert Gray code values to their binary equivalents. An understanding of these numbers and their applications can benefit the machine language programmer and can improve code generation in compilers. The applications for these constants will arise naturally in the construction of algorithms for the solution of each problem.

Introduction

The field of mathematics has uncovered many "magic numbers." These are numbers which have special or unusual properties when used in certain calculations. For example, the magic number 142,857 is the smallest "cyclic number."[1] Cyclic numbers produce a result that is a cyclic permutation of the original number's digits when multiplied by an integer less than or equal to the number of digits in the original number:

```
1 * 142,857 = 142,857
2 * 142,857 = 285,714
3 * 142,857 = 428,571
4 * 142,857 = 571,428
5 * 142,857 = 714,285
6 * 142,857 = 857,142
```

Most magic numbers have little or no application apart from being mathematical curiosities, and no computer uses are known for them. Cyclic numbers are no exception to this; they are elegant but have no practical value. One of the reasons for this is that many numerical tricks like cyclic numbers rely on decimal (base 10) format. Most computers operate uncomfortably at best using base 10 arithmetic and prefer to stick to binary (base 2). Most exploration for number "magic" has been conducted in the decimal realm, so there has been only limited discussion of binary magic numbers.

Binary magic numbers do exist, and these numbers can be used to significantly enhance certain computer operations. This article concentrates on a specific sequence of binary magic numbers. The Nth number in this sequence will consist of an infinite sequence from right to left of 2^N 1's, followed by 2^N 0's, followed by 2^N 1's, and so on:

```
. . . 0101010101010101
. . . 0011001100110011
. . . 0000111100001111
. . . 0000000011111111
                    :
                    :
```

In practice, however, binary operations are limited to a finite word size. For a word size of 8 bits, there will be three numbers:

```
B[1] := 01010101
B[2] := 00110011
B[3] := 00001111
```

There will be four B-constants for word sizes of 9-16 bits, five for sizes of 17-32 bits, and so on. The bit complements of these binary values will also prove to be useful:

```
B[4] := 10101010
B[5] := 11001100
B[6] := 11110000
```

Notation

All of the example algorithms in this article are presented in Pascal. The reader is assumed to be familiar with Pascal and with the basics of boolean algebra.

Pascal does lack full-word logical operations (logical functions which apply on a bit-for-bit basis to each bit in an integer value) as standard functions, so the following special functions are assumed to be implemented:

IAnd (X, Y) — return bit-for-bit AND of X and Y.

IOr (X, Y) — return bitwise inclusive OR of X and Y.

IXor (X, Y) — return bitwise exclusive OR of X and Y.

INot (X) — return bit-for-bit complement of X.

LShift (X, N) — return X shifted left by N bit positions. Zeroes are shifted into the low-order bits.

RShift (X, N) — return X shifted right by N bit positions. Zeroes are shifted into high-order bits.

Some of the methods shown assume integer overflows will not be detected. No other extensions to Pascal are used. All variables are explicitly declared and the integer type is the only type of variable used. Some global machine-dependent constants are assumed to have been declared outside the example routines. N is the number of bit positions in an integer value. V is the base 2 logarithm of N rounded to the next highest integer. The values B[1] through B[2*V] are considered to be global constants and are frequently used in the routines. B[1] through B[V] will be the binary magic numbers defined above and B[V+1] through B[2*V] will be their bit-for-bit complements. The constants S[1] through S[N] are a table of the powers of 2 within the range of the machine. S[Q] will be equal to 2^Q. The global constants might be initialized on a 12-bit machine by th Pascal code fragment in Figure 1, below.

Most of the algorithms will be more useful if they are recoded in a specific machine language. No sophisticated Pascal constructs like sets or records are used, so the translation process should be quite

```
const
  N = 12; V = 4;
var
  B : array [1..8] of integer;
  S : array [1..12] of integer;
  I : integer;
begin
  B[1] := 010101010101; B[2] := 001100110011;
  B[3] := 111100001111; B[4] := 000011111111;
  B[5] := 101010101010; B[6] := 110011001100;
  B[7] := 000011110000; B[8] := 111100000000;
  for I := 1 to N do S[I] := LShift (1, pred (I));
  ...
```

Figure 1.

easy. The special logical functions will correspond to single instructions on most machines.

Some of the algorithms discussed below will involve the positions of bits within words. Some consistent numbering scheme for these bits is required. The least significant bit of a value is called bit 0 and numbering proceeds to the left. The sign bit of the value (most significant bit) will be bit (N-1). If the eight digit positions in an 8-bit word were each filled with their corresponding numbers, the result would be 76543210. Note that this numbering scheme can be derived from the constants B[V+1] through B[2*V] if they are read as binary values from bottom to top in each digit column. This is a very important property of these numbers that will be used frequently.

A few of the algorithms assume that negative values are stored in two's-complement form. This is usually not a problem – there are not very many one's-complement or signed-magnitude machines.

Locating Bits

One commonly needed binary operation is the ability to locate the lowest or highest zero or nonzero bit in a value. This is often used in operating systems, when sets of boolean flags are grouped together in a single word and each flag requires a specific action. Bit location operations are of particular interest in scheduling algorithms, where quick execution is essential.

Many large machines include special "exotic" instructions to locate zero or nonzero bits in a value. The DEC-10 has the JFFO instruction, which is a mnemonic for "jump if found first one."[2] This operation tries to find the highest nonzero bit in a 36-bit value and optionally branches if the operation is successful. By contrast, the VAX-11 instructions set includes the FFS and FFC instructions, which are mnemonics for "find first set" and "find first clear," respectively.[3] These instructions search a field (an extracted portion of a 32-bit value) from the lowest bit up for the first zero or nonzero bit. Neither machine includes an explicit instruction for searching in the opposite direction, although the CVTLD (convert long integer to double-precision floating) instruction on the VAX-11 might be used for this. It is also interesting to note that the instruction execution speed is bit-position dependent on the DEC-10 (the farther away from an 18-bit half-word boundary, the longer the instruction takes) and is position independent on the VAX-11 (the same amount of time is taken for each bit position).[4] It is vital to be aware of such speed dependencies if exotic instructions like these are used.

Smaller machines do not usually include such special instructions. The most obvious way to perform this operation is to successively shift the value in question until a bit in the proper state is found. The lowest bit can be checked with an AND, and the highest bit can be checked by watching for a sign change. The functions in Figure 2, page 28, illustrate these methods.

These functions return a result of -1 if no bit that satisfies the specified condition can be found, and the bit position of the selected bit otherwise. Up to N iterations within the functions may be necessary to find the selected bit. However, it is possible to make use of the B-constants and check more than one bit position with each iteration. The functions in Figure 3, page 28, perform complementary operations to the functions shown above by making use of the B-constants. Note that the LowClearBit function operates on the complement of the input value and not on the value itself.

The functions operate on multiple bit fields at once. Each iteration of the **for** loop isolates the desired bit to one-half of the remaining bits in a word. The weighting of the bit positions is used to advantage in computing the final position by shifts and adds instead of using a table of constants like S. The S-constants could be used if the shift operation is difficult to implement. The assignment of Result within the **for** loop is superfluous during the first iteration and may be skipped.

These algorithms take V iterations in all cases. This can be far less than N for large word sizes, but the iterative loop operation is more complex. The distribution of bits in the input values that appear in a given application must also be considered. The LowSetBit function presented in Figure 2 may use an average of fewer than V iterations if nonzero low-order bits are prevalent in the input values.

It is possible to simplify the last two functions if it is known that there is only a single nonzero bit in the value. It becomes unnecessary to replace the input value as the algorithm proceeds since the desired bit is already isolated, so the Temp and Ival variables are not needed.

It is very easy to check for a single nonzero bit in a word. Given Value, "((Value <> 0) **and** (IAnd (Value, pred (Value)) = 0))" will be true if and only if there is a single nonzero bit in Value.[5] An alternate test is "((Value <> 0) **and** (IAnd (Value, -Value) = Value))."

Sideways Addition

The sideways sum of a number is the sum of all the individual digits in the number taken one at a time. The sideways sum will be different depending on the base the number is represented in. For example, the sideways sum of the decimal value 12345 is 15, and the sideways sum of the binary value 10110 is 11 (in binary). A binary sideways sum is just the count of the 1's in the binary value.

Sideways sums have many interesting mathematical properties and applications. Decimal sideways sums are the basis for an old trick for determining divisibility by 3 or 9. If a number is evenly divisible by 3 then the decimal sideways sum will also be evenly divisible by 3. The same property holds true for 9 as well. (The decimal sideways sum always gives a value with the same remainder modulo 3 or 9 as the original value.) Since the sideways sum of a number is usually much smaller than the number itself, it is easy to check divisibility of a large number by iteratively computing sideways sums until a manageably small value is obtained.

A binary value can be thought of as a representation of a set. Each digit corresponds to a single element that may possibly be in the set. A digit value of 0 means the element is not present and a value of 1 means the element is present. In this scheme (which is used by many programming languages), the binary sideways sum indicates the cardinality of a set – it is a count of the number of elements in the set.

Some machines have implemented the binary sideways sum as an instruction. The CDC (Control Data Corporation) 6400, 6500, and 6600 series processors implement the "CXi Xk" instruction. This instruction counts the number of nonzero bits in register Xk and stores the result in register Xi. i and k are single octal digits and serve to identify one of the eight 60-bit X registers available on the machine. The time used for this operation varies even within this series of machines; the 6400 and 6500 require 68 minor cycles while the 6600 needs only eight. For comparison purposes, an integer addition takes six minor cycles on the 6400 and 6500 and three minor cycles on the 6600.[6]

The most obvious way to compute a binary sideways sum is to iteratively shift the value right and count the low-order bits as shown in Figure 4, at left.

This method can take up to N iterations, but it typically takes fewer than N since small values are fairly common. There are many possible variations on this basic type of function, including using bits shifted out the top of a value, shifting an entire field out of the value and looking up its sideways sum in an auxiliary table, and using floating-point normalization hardware if it is available.

The test for a single nonzero bit presented above may also be adapted to produce a sideways summation algorithm (see Figure 5, at left). The number of iterations for this method is equal to the final sideways sum result. The average number of iterations would be N/2 if a random value (each bit has an equal

```
function LowSetBit (Value : integer) : integer;
var
  Temp, Result : integer;
begin
  if Value = 0 then LowSetBit := -1
  else begin
    Temp := Value;
    Result := 0;
    while not odd (Temp) do begin
      Temp := RShift (Temp, 1);
      Result := succ (Result);
    end;
    LowSetBit := Result;
  end;
end;

function HiClearBit (Value : integer) : integer;
var
  Temp, Result : integer;
begin
  Temp := Value;
  Result := pred (N);
  while Temp < 0 do begin
    Temp := LShift (Temp, 1);
    Result := pred (Result);
  end;
  HiClearBit := Result;
end;
```

Figure 2.

```
function LowClearBit (Value : integer) : integer;
var
  Temp, Result, Ival, I : integer;
begin
  if Value = -1 then LowClearBit := -1
  else begin
    Ival := INot (Value);
    Result := 0;
    for I := V downto 1 do begin
      Result := LShift (Result, 1);
      Temp := IAnd (Ival, B[I]);
      if Temp = 0 then Result := succ (Result)
      else Ival := Temp;
    end;
    LowClearBit := Result;
  end;
end;

function HiSetBit (Value : integer) : integer;
var
  Temp, Result, Ival, I : integer;
begin
  if Value = 0 then HiSetBit := -1
  else begin
    Ival := Value;
    Result := 0;
    for I := V downto 1 do begin
      Result := LShift (Result, 1);
      Temp := IAnd (Ival, B[V+I]);
      if Temp <> 0 then begin
        Result := succ (Result);
        Ival := Temp;
      end;
    end;
    HiSetBit := Result;
  end;
end;
```

Figure 3.

chance of being 0 or 1) is input.

It is possible, however, to operate on multiple fields within a value simultaneously, first adding adjacent bits, then bit pairs, and so on. The coding of this method is set forth in Figure 6, at left. This method uses only the uncomplemented B-constants and takes V iterations in all cases. There is only a slight increase in the number of operations required in the loop over the first SideSum function. This method is preferred when a large number of 1's are expected in the input value.

Parity

A slight variation on sideways addition is to just record whether or not the sum of the digits is even or odd. This result is called the parity of a number and applications for it frequently arise in error correction techniques.

Parity is usually thought of as the exclusive-OR of all the binary digits in a value – thus a 0 indicates even parity and a 1 indicates odd parity. Many microcomputers include hardware to compute parity bits easily, but many larger machines do not have such facilities. The simplest way to compute parity, illustrated in Figure 7, at left, is analogous to the first method for sideways sum computation involving repetitive shifts and tests.

As before, this operation can take up to N iterations but typically requires somewhat fewer. The method can be altered to take advantage of link bits or floating-point normalization instructions (which typically shift the uppermost nonzero bit to the highest bit position automatically). For example, the 8087 numeric data processor can be used in this fashion by reading an integer value into the unit and subsequently writing the value out in floating-point form.[7] The mantissa of the result will then be the original integer shifted left so that the highest nonzero bit is at the top of the field, and the exponent is an indication of the number of shifts that were needed. Care must be taken to intercept negative values and to use a floating-point format capable of holding an entire integer in its mantissa without loss of accuracy. The 8087 is slightly cumbersome to use in this fashion as it requires reading and writing the values into memory instead of performing the operation entirely in registers. Machines with simple conversion instructions are actually more versatile in this case.

The second method presented for sideways addition can also be adapted to compute parity. This algorithm is easily derived from the SideSum function and will not be presented here.

Figure 8, at left, shows how the third method for sideways addition can be adapted into a parity function with considerable ease. This algorithm always takes V iterations. The B-constants are not re-

quired since exclusive-OR operations do not generate carries and masking is not necessary.

One peculiarity that must occasionally be dealt with is machines that do not possess exclusive-OR hardware. Of course, the exclusive-OR may be coded in terms of other logical operations (see Figure 9, at left). This is a very costly operation. It may be more practical to compute the full sideways sum and extract the lowest bit rather than using this cumbersome function.

The third sideways sum technique may alternately be adapted to provide the parity of a value using only ANDs, shifts, and normal addition. This is accomplished by noting that an exclusive-OR is essentially an addition with no carry, so all that must be done is to prevent additive carries from crossing from one field of the number to the next, as in Figure 10, at left. This technique is interesting in that it requires only one of the binary magic numbers. (V-1) iterations are required, but some preliminary calculations of greater complexity than the iterative one must be performed.

Another peculiarity of some machines is the lack of an AND instruction. Such machines, notably the LSI-11, PDP-11, and VAX-11, have a bit test instruction (BIT) that performs an AND but does not store the result. These machines also have a bit clear instruction (BIC), which corresponds to an "AND-NOT" operation.[3] This does not present a problem with any of the algorithms shown in this article. Most uses of IAnd involve a constant. The constant is just complemented and then the BIC operation can be used. One exception to this is the IXor function, which can be recoded more simply using the statement "IXor := IOr (BIC (X, Y), BIC (Y, X))." The only other exceptions are the tests for a single nonzero bit. The first test does not require the storage of the result, so a BIT instruction could be used or an INot can be inserted. The second test can be expressed as "((Value <> 0) **and** (BIC (Value, pred (Value)) = Value))."

Weighted Sideways Addition

Sometimes it is necessary to perform sideways addition with each digit weighted differently. This requires multiplying each digit by a fixed constant before summation is done. The way numbers are normally interpreted is merely a special case of this operation; each digit of a number is weighted by successive powers of the base of the number system. Thus numbers themselves can be thought of as just collections of digits where a particular type of weighted sideways addition is used to compute their value. The conventional sideways sum is another special case – each digit has unary weight.

```
function SideSum (Value : integer) : integer;
var
  Temp, Result : integer;
begin
  Temp := Value;
  Result := 0;
  while Temp <> 0 do begin
    Result := Result + IAnd (Temp, 1);
    Temp := RShift (Temp, 1);
  end;
  SideSum := Result;
end;
```

Figure 4.

```
function SideSum (Value : integer) : integer;
var
  Temp, Result : integer;
begin
  Temp := Value;
  Result := 0;
  while Temp <> 0 do begin
    Result := succ (Result);
    Temp := IAnd (Temp, pred (Temp));
  end;
  SideSum := Result;
end;
```

Figure 5.

```
function SideSum (Value : integer) : integer;
var
  Result, I : integer;
begin
  Result := Value;
  for I := 1 to V do
    Result := IAnd (RShift (Result, S[I]), B[I]) +
      IAnd (Result, B[I]);
  Sidesum := Result;
end;
```

Figure 6.

```
function Parity (Value : integer) : integer;
var
  Temp : integer;
begin
  Temp := Value;
  Result := 0;
  while Temp <> 0 do begin
    Result := IXor (Temp, Result);
    Temp := RShift (Temp, 1);
  end;
  Parity := IAnd (Result, 1);
end;
```

Figure 7.

```
function Parity (Value : integer) : integer;
var
  Result, I : integer;
begin
  Result := Value;
  for I := 1 to V do
    Result := IXor (Result, RShift (Result, S[I]));
  Parity := IAnd (Result, 1);
end;
```

Figure 8.

```
function IXor (X, Y : integer) : integer;
begin
  IXor := IAnd (INot (IAnd (X, Y)), IOr (X, Y));
end;
```

Figure 9.

```
function Parity (Value : integer) : integer;
var
  Result, I : integer;
begin
  Result := IAnd (IAnd (RShift (Value, 1), B[1]) +
    IAnd (Value, B[1]), B[1]);
  for I := 2 to V do
    Result := IAnd (RShift (Result, S[I])
     + Result, B[1]);
  Parity := IAnd (Result, 1);
end;
```

Figure 10.

```
function WeightedSum (Value : integer) : integer;
var
  Temp, Result, I : integer;
begin
  Result := 0;
  Temp := Value;
  I := 1;
  while Temp <> 0 do begin
    if odd (Temp) then Result := Result + C[I];
    Temp := RShift (Temp, 1);
    I := succ (I);
  end;
  WeightedSum := Result;
end;
```

Figure 11.

```
function WeightedSum (Value : integer) : integer;
var
  Temp, Result, I : integer;
begin
  Result := 0;
  for I := 1 to K do begin
    SideSum (IAnd (Value, D[I]), Temp);
    Result := LShift (Result, 1) + Temp;
  end;
  WeightedSum := Result;
end;
```

Figure 12.

Weighted sideways sums can be used to generate arbitrary mappings and permutations of bits within words. A permutation is a rearrangement of the bits within a word where no bits are lost, and a mapping is a more general operation in that a bit may be placed in more than one position in the final result and certain bits may be lost. It is trivial to construct a set of weights for a sideways sum that will accomplish any mapping or permutation – each bit's weight is a mask containing 1's in the bit positions the bit is to be mapped into.

The additive properties of a weighted sideways sum can be used in a more general sense, however. Consider a binary value as a representation of a set, as it has been discussed earlier. Each potential element of the set might be said to have certain properties. Different elements will have different properties. There can be many properties associated with each member. Take the set of fruits [LIME, LEMON, ORANGE]. One property might be color – [GREEN, YELLOW, ORANGE]. Another might be cost – [41 cents, 35 cents, 52 cents]. A third might be ordinal position – [first, second, third].

Some properties are not additive (like color) and others (position or cost) are. It may not make sense to total the positions of the elements of a set, but a property like cost might be useful to sum over an entire set. In the binary representation of a set, the costs could be thought of as weights and the resultant sideways sum as the cost of the entire set.

The straightforward way to compute a weighted sideways sum is to maintain a table of constant weights and to iteratively shift the input value. For the function shown in Figure 11, page 33, a table of constant weights C[1] through C[N] applying to each respective bit position is assumed to be defined.

This routine takes a maximum of N iterations, which is to be expected. There is no clear way to write a more efficient routine using the B-constants. It is necessary to examine the constant weights in C for certain bit patterns. Consider the following ascending sequence for an 8-bit machine:

C[1] := 0001	C[5] := 0101
C[2] := 0010	C[6] := 0110
C[3] := 0011	C[7] := 0111
C[4] := 0100	C[8] := 1000

A new set of weights can be derived by reading the columns of the C array. This is equivalent to constructing an 8 x 8 matrix with the rows of C and taking the reversed rows of the transpose of this matrix. This will create a new set of constants D[1] through D[K]. K = 4 for the example C-constants given above, since D[5] through D[8] will all be zero. In this example, the D-constants are actually shifted versions of the B-constants.

```
function ReverseBits (Value : integer) : integer;
var
  Temp, Result, I : integer;
begin
  Temp := Value;
  for I := 1 to N do begin
    Result := LShift (Result, 1) + IAnd (Temp, 1);
    Temp := RShift (Temp, 1);
  end;
  ReverseBits := Result;
end;
```

Figure 13.

```
function ReverseBits (Value : integer) : integer;
var
  Result, I : integer;
begin
  Result := Value;
  for I := 1 to V do
    Result := IAnd (LShift (Result, S[I]), B[V+I]) +
      IAnd (RShift (Result, S[I]), B[I]);
  ReverseBits := Result;
end;
```

Figure 14.

```
function GrayToBinary (Value : integer) : integer;
var
  Temp, Result : integer;
begin
  Result := 0;
  Temp := Value;
  while Temp <> 0 do begin
    Result := IXor (Result, Temp);
    Temp := RShift (Temp, 1);
  end;
  GrayToBinary := Result;
end;
```

Figure 15.

```
function GrayToBinary (Value : integer) : integer;
var
  Result, I : integer;
begin
  Result := Value;
  for I := 1 to V do
    Result := IXor (Result, RShift (IAnd (Result,
      T[I]), S[I]) *
      (pred (S[succ (S[I])])));
  GrayToBinary := Result;
end;
```

Figure 16.

Now the D-constants can be used to reduce the weighted summation to a series of unweighted sideways sums. Each unweighted sum builds a single quantity of constant weight for the final result, as illustrated in Figure 12, below.

This method is by no means faster than the conventional version of WeightedSum for a general set of weights. However, certain special cases (such as the example weights given in Figure 12) will introduce a multitude of simplifying reductions if the SideSum function is coded inline, particularly if the D-constants bear some resemblance to the B-constants. The reduction of the example is left as an exercise to the reader. Which WeightedSum method takes fewer operations?

Bit Reversal

Another interesting operation to study is that of reversing all the bits in a word. Bit 0 will become bit (N-1), bit 1 will become bit (N-2), bit 2 will become bit (N-3), and so on. This is actually a special case of the weighted sideways sum operation, since it is nothing more than a permutation of the bit positions in a value. One application for bit reversal has already been encountered in the construction of the D array from the C-constants for the second method of weighted sideways addition.

Bit reversal may easily be implemented by a series of shifts (see Figure 13, below). N iterations are always required for this function.

This function may be modified to reverse the bits in a quantity (2*N) bits long. The high-order part of the input quantity is passed in Result, and all that is necessary is to modify the function to copy the sign bit of Result to the sign bit of Temp on each iteration. The final value will be contained in Temp and Result. No additional iterations are needed.

Many small computers have an additional bit which is attached to the high-order end of the register used for shift operations. This bit is usually called the link or carry flag. This bit can be a nuisance when performing some circular rotate operations since it acts as an extra bit position between the least significant bit and the sign bit. Nevertheless, a carry flag can be used to advantage in bit-reversal routines. For example, the Motorola 6800 can be used for bit reversal of a 16-bit quantity if the high-order 8 bits of the input value are placed in accumulator A, the low-order 8 bits of the input value are placed in accumulator B, the following two instructions are repeated eight times:

```
ROL A
ROR B
```

and finally a single "ROL A" instruction is executed. The result will be in the accumulators in the same configuration as the input value. The H, I, and C flags will be unaffected and the N, Z, and V flags will

be set according to the high-order 8 bits of the result.

It is possible to use the B-constants to minimize the number of iterations required as in Figure 14, at left. This method is restricted to machines where N is a power of 2. This will not be a problem on most processors since word lengths of 8, 16, and 32 bits are the most common. There are some processors, however, where this method cannot be implemented, like the PDP-8 and Intersil 6100 (12-bit), the DEC-10 (36-bit), and most CDC machines (60-bit). This method requires only V iterations in all cases. It should be avoided on machines which have difficulty in shifting a quantity by more than one bit position at once, and on machines whose word length is not a power of 2.

Gray Codes

Gray codes are essentially an alternate set of counting numbers that have different rules than most conventional number systems. When Gray code numbers are placed in counting order, any adjacent pair of numbers will differ in at most one digit position, and the absolute value of the difference in that digit position will always be one.

There are an infinite number of Gray codes, and Gray codes exist for every numeric base. The simplest Gray code of all is the binary reflected Gray code, which will be the code under discussion here. The binary reflected Gray code is used more often than any other Gray code.

Gray codes have many applications such as error correction in pulse-code modulated systems. There are several places in recreational mathematics where Gray codes turn up; the disk motion of the Towers of Hanoi problem is an example.[8]

It is easy to convert an ordinary binary number into its binary reflected Gray code equivalent. If Value is the binary number, "IXor (Value, RShift (Value, 1))" will be the Gray code equivalent for Value. The conversion of a Gray code number to its binary equivalent is not so easy. Each bit of the original binary number may be recreated by taking the exclusive-OR of the corresponding bit in the Gray code with all the bits to its left (higher-order bits including the sign bit). For example, a binary value of 0111 would be converted to 0100 in *Gray code.*

The conversion of a Gray code value into its original binary form may be accomplished by the function in Figure 15, page 34. The function requires at most N iterations to complete the operation.

The final value created can be represented in tabular form as an "addition problem," where exclusive-OR is used instead of normal addition. If each bit position in Value is labeled with a letter "a" through "n" (assuming that N=8 and "a" is the label for the highest bit), the problem can be seen as:

```
  abcdefgh
   abcdefg
    abcdef
     abcde
      abcd
       abc
        ab
         a
--------------
  Result
```

In order to solve the problem in V iterations, it is useful to first define the set of augmented constants T[1] through T[V] by the relationship "T[Q] := IAnd (LShift (B[Q], 1), LShift (B[Q], S[Q]))." The T-constants are shown here for N = 16:

```
T[1] := 1010101010101010
T[2] := 0100010001000100
T[3] := 0001000000010000
T[4] := 0000000100000000
```

The Nth T-constant is formed by repeating a pattern from right to left of (2^{N-1}) Os, followed by a single 1, followed by $(2^{N-1}-1)$ Os.

The method used with the T-constants is vaguely similar to the "divide and conquer" method of Ercegovac.[9, 10] Each iteration builds several different fields of the final result. Multiplication is used in the loop – this method should be avoided on machines which do not have integer multiplication hardware. The multiply operation is used to shift single-bit fields into multiple positions in the word (see Figure 16, page 35).

The pattern generated by this function is quite interesting. After the first iteration, the value in Result is (using the same "addition problem" notation as above and assuming N = 8):

```
  abcdefgh
   a c e h
--------------
  Result
```

After the second iteration, the pattern is:

```
  abcdefgh
   abc efg
    ab  ef
     a   e
--------------
  Result
```

On the third and final iteration, the binary equivalent is obtained.

Generation of Constants

It may be necessary in some programming environments to create the B-constants dynamically instead of just reading them from a table. The B-constants are easily generated using the S-constants (which are in turn quite easy to create using a single LShift). See Figure 17, at right.

The dynamic generation of the B-constants may be "blended" with the inner loop of most of the algorithms given in this article. Each iteration of the loop then generates the constant it requires using a temporary variable that is initially set to -1. Unfortunately, it does not appear to be possible to perform this operation in the reverse order, that is, derive B[Q+1] from B[Q]. This makes it difficult to add the code for dynamic generation of the B-constants to the inner loop of the SideSum function. Although the ReverseBits function loop is written in ascending order, it may operate in either direction, so it is possible to generate the B-constants as the loop proceeds.

The T-constants used in the GrayToBinary algorithm may be generated from the B-constants by using their defining relationship. Alternately, a similar iterative process can be used, which also proceeds in the wrong direction to make it useful within the inner loop of the GrayToBinary function (see Figure 18, page 36). The assignment of Temp in the last iteration of the loop is not needed and always sets Temp to -1.

Advanced Applications

More sophisticated applications for the B-constants exist. They can be used to transpose bit matrices[11, 12] and there are even applications for these constants in certain error correcting codes. These applications are beyond the scope of this article, and might be the subject of a future one.

Conclusion

This article has covered many applications for this particular set of binary constants. The behavior of all these algorithms seems to have a lot of things in common – the normal and obvious method for performing the operation typically takes up to N iterations, where N is the word size of the machine, and the method using the B-constants takes V operations, where V is the base 2 logarithm of N rounded up to the nearest integer. For large word sizes (or when operating on multiword values with a smaller machine), these algorithms represent a considerable savings in execution time. Judicious use of these techniques may improve many machine-language programs.

Acknowledgements

I wish to thank Donald Knuth, whose work provided the basis of many of the ideas expressed in this article.[11] I also wish to thank Douglas Grover and Kevin Carosso of the MATHLIB project, who provided the problems to solve that led to this article along with many helpful suggestions.

```
procedure CreateBConstants;
var
  Temp, I : integer;
begin
  Temp := -1;
  for I := V downto 1 do begin
    Temp := IXor (Temp, LShift (Temp, S[I]));
    B[I] := Temp;
    B[V+I] := INot (Temp);
  end;
end;
```

Figure 17.

```
procedure CreateTConstants;
var
  Temp, I : integer;
begin
  Temp := 1;
  for I := V downto 1 do begin
    T[I] := LShift (Temp, S[I]);
    Temp := IOr (Temp, T[I]);
  end;
end;
```

Figure 18.

References

1 M. Gardner, "Cyclic Numbers and Their Properties," *Scientific American*, March 1970, pp. 121-124.

2 *DECsystem-10 Assembly Language Handbook*, Digital Equipment Corporation, 1972, pp. 2-56 through 2-57.

3 *VAX-11 Architecture Handbook*, Digital Equipment Corporation, 1982.

4 N. Freed, K. Carosso, and D. Grover, *MATHLIB Project Software Library User's Guide*, Harvey Mudd College, 1982.

5 D. Cortesi, "Dr. Dobb's Clinic – DUP DUP NEG AND =," *Dr. Dobb's Journal*, Vol. 6, No. 9, p. 60.

6 C. Redeker and R. Duncan, *COMPASS – An Assembly Language*, University of Washington, Seattle, Washington, 1977.

7 R. Duncan, "Intel's 8087 Numeric Data Processor," *Dr. Dobb's Journal*, Vol. 7, No. 8, pp. 47-50.

8 M. Gardner, "The Curious Properties of Gray Code and How It Can Be Used to Solve Puzzles," *Scientific American*, August 1972, pp. 106-109.

9 M. Ercegovac, "A Fast Gray to Binary Conversion," *IEEE Proceedings*, April 1978, pp. 524-525.

10 D. Agrawal, "A Comment on Gray to Binary Conversion," *IEEE Proceedings*, March 1979, pp. 444-445.

11 D. Knuth, *The Art of Computer Programming – Combinatorial Algorithms*, Vol. 4 (preprint version), Stanford University, Stanford, California.

12 L. Guibas and J. Stolfi, "A Language for Bitmap Manipulation," *ACM Transactions on Graphics*, Vol. 1, No. 3, pp. 192-214.

DDJ

An 8080 Fig-Forth Directory and File System

This article describes a Forth program which overlays a directory and file system on the version of Fig-Forth which has been adapted to the North Star disk system by Interactive Computer Systems, Inc., 6403 Demarco Road, Tampa, Florida 33614. The program requires only one CODE word linkage to the North Star DOS to implement a dozen useful file commands. Except for this one disk-system-peculiar word, the entire file overlay is written in high-level Forth; hence the end product is highly portable.

The article commences with a commands summary in Fig-glossary style. This is followed by a short discussion of the author's rationale for the approach used and a description of the North Star disk and directory formats. The article concludes with a few explanatory notes as an aid in deciphering the appended listing.

Summary of Commands

All commands that prompt for input can be aborted after the prompt occurs by typing a ctrl-c. Input errors which are detected by the operator, as opposed to the command itself, may be corrected by typing a space and re-entering the correct input after the prompting error message. File names and disk names may be from one to eight of any printing ASCII characters from ! to Z inclusive (33 hex to 5A hex inclusive). Note that this differs from the North Star format, which requires alpha and numbers only. The validity of the range of drive numbers (1-3) is checked in all commands where drive number is an input.

The following is a summary of the commands in Fig-glossary style:

CD DR#1 DR#2 . . .

Copies diskette on drive DR#1 to diskette on drive DR#2.

CF DR#1 DR#2 . . .

Copies a named file on drive DR#1 to a named file on DR#2. The names of the files are prompted. The file on drive DR#2 does not have to pre-exist; its *directory* entry will be created. If it does pre-exist, the file transfer will occur only if the file is sufficiently large. If the file is created, it will be checked for possible disk overflow. If overflow would occur, the command aborts.

CLR-SCR SCR#1 SCR#2 . . .

Clears a block of screens from SCR#1 to SCR#2.

CRF DR#1 . . .

Creates a named file on drive DR#1. Prompts for file name, screen start, screen stop, file type. Will create a directory entry for all file types except a Forth data file (type 5). Forth source program files should be designated as type 4. Type 1 files (object code) will request a hex load-and-go address. In keeping with the general permissiveness of Forth, no check is made for the validity of the screen numbers. Checks for existing file of same name and aborts if found. Existing file entry may be changed by ED command followed by CRF command. Does not check for potential disk overflow and assumption is that all screens of named file are contiguous on the same disk.

CSCR SCR#1 SCR#2 SCR#3 . . .

Copies block of screens SCR#1 to SCR#2 inclusive to block of screens starting at SCR#3.

CSEC SEC#1 SEC#2 DR#1
SEC#3 DR#2 . . .

Copies block of sectors SEC#1 through SEC#2 on drive DR#1 to the block starting at sector SEC#3 on drive DR#2. *Warning*: No check is made to check the block start address SEC#3 for potential overlap with existing named files.

DE DR#1 . . . .

Deletes the specified named directory entry on the specified drive DR#1. File name is prompted. Deletes the directory entry only. Does not access the disk file region in any way; consequently this command may be used to change the file parameters (see CRF command).

FILE #SCRNS DR#1 FILE
FILE-NAME

BYTE# DR#1
FILE-NAME

At definition time, FILE creates a named type 5 Forth data file in the directory of the specified drive. When FILE-NAME is executed, the stack will contain the RAM address of the specified byte on the stack. At definition time, a check is made for potential disk overflow. The command aborts on such an eventuality. Likewise, a pre-existing file of that name on the drive will abort the file definition, i.e., duplicate file names on the same drive are not permitted. Insufficient directory space also aborts the command. *Caution*: No check is made on the range of the specified byte. The purpose of this word is to allow disk data files to be created and manipulated in Forth source programs. Files created using this word cannot be accessed by the LIST-FILE and LOAD-FILE commands. (An example using FILE is shown in Figure 1 on this page.)

LI DR#1 . . . .

Lists the directory of the specified drive in North Star format, that is, file name, sector start, file length in sectors, file type, and load-and-go address if file is of type 1.

LI.S DR#1 . . . .

Lists the directory of the specified drive in the format file name, screen start,

by Alfred J. Monroe

Alfred J. Monroe, 3769 Grandview Blvd., Los Angeles, California 90066.

```
( create a file 10 screens long on drive #2 )
          10 2 FILE TST-FILE
 ( write ASCII characters into the file )
: WRT 90 33 DO      ( from ASCII ! to Z )
        500 I +       ( form byte address 500+I )
        2             ( drive # )
        TST-FILE    ( invoke file RAM addr )
        I             ( ASCII value to stack )
        SWAP C!     ( store into file )
        LOOP        ( keep going till Z )
        FLUSH       ( be sure file goes to the disk )
;
( read the ASCII characters out to the terminal )
: READ 90 33 DO 500 I + 2 TST-FILE C@ EMIT LOOP ;
```

Figure 1.

screen stop, file type, and load-and-go address if file is type 1.

LIST-FILE DR#1 . . . .

Used to list the contents of a Forth source program file. Prompts for the name of the file. Checks file type. File type must be 4.

LOAD-FILE DR#1 . . . .

Used to load a named Forth source program file from the specified drive. File type must be 4.

ND DR#1 . . . .

Provides a means of uniquely identifying each diskette as the first entry in the file directory. It reserves the first four sectors of the disk for the directory. This command will overwrite any pre-existing first *entry. Prompts* for the disk name.

Approach

Several of the commands described above prompt the operator for additional parameters. Some are forgiving in the sense that they provide for error recovery in the event of a syntax error. In the event that the error is of the type that is not caught by the routine, e.g., input of a decimal number that is unintentional, simply enter a space and proceed to enter the correct sequence of digits after the error message. It is for this reason that (word) and (number) are not used in the routines. In addition, the error messages have been made a part of the definition to avoid the necessity of maintaining the master diskette in drive one, as would be the case if "message" had been used.

The North Star DOS provides a number of useful routines through a jump table at the beginning of DOS:

DLOOK	searches directory for a file name.
DWRIT	writes directory back to disk.
DLIST	lists the directory.
DCOM	reads or writes to disk.

In the initial approach to a file system development, maximum use was made of these routines. This approach was ultimately abandoned principally because it severely reduced the portability of the end result. The file system presented here is written entirely in high-level Forth with the exception of the READ/WRITE to the disk. This function is accomplished by means of code word linkage to DCOM. As a result, the current design should prove useful on systems other than the North Star disk system.

Description of Disk and Directory Formats

The North Star disk format consists of 35 tracks of 10 sectors per track for a total of 350 sectors per diskette. They are numbered from 0 to 349 inclusive. Each sector consists of 256 bytes. The first four sectors (= 1024 bytes = 1 screen) are reserved for the directory. There is a maximum of 64 entries. Each entry consists of 16 bytes. Bytes 0-7 are the name made up of from one to eight of any printing ASCII characters and filled with trailing blanks for names less than eight characters long. Note that this generalizes the name convention of North Star. Bytes 8-9 (lo-hi) are the binary decimal value of the disk sector start address of the file. Bytes 10-11 (lo-hi) are the binary decimal value of the length of the file in sectors. Byte 12 is the binary value of the file type. North Star has defined 0 as the default type, 1 for executable object code, 2 for BASIC source programs, and 3 for BASIC data files.

This Forth file system further defines a Forth source program file as type 4, and a Forth data file as type 5. Bytes 13-15 are type dependent. In particular, for a file of type 1, bytes 13-14 (lo-hi) are the hexadecimal RAM address of file-load and start-of-execution, and byte 15 is unused.

The North Star disk and directory formats are adhered to in this Forth file system so that disk directories may be read and utilized in DOS and BASIC. Since Fig-Forth assumes 512 bytes per block, two blocks per screen, manipulation of a directory entry must observe the conversion factors that there are two sectors per block and four sectors per screen.

Deletion of a directory entry is accomplished by replacing the entry with sixteen blanks. The disk file is left untouched; that is, it is recoverable in the event of accidental erasure of its directory entry. Similarly, creation of an entry does not alter the disk elsewhere; hence increasing the length of a file or changing other file descriptive parameters may be accomplished by erasing its name from the directory and recreating the file name with the new file parameters.

Directory entries for Forth source files are created by operator keyboard input of screen start and screen stop. It is assumed that such source files are a set of contiguous screens on the same disk. In keeping with the general permissiveness of Forth, no check is made that the screen numbers are valid. However, all screen numbers are converted to the proper sector numbers.

On the other hand, creation of a Forth data file is inhibited if its length would cause a disk overflow, and the copy file command (CF) will also abort if it would lead to disk overflow.

The fact that the directory is in terms of file sector start and file sector length is not very convenient for use in Forth. Therefore, two ways of listing the directory have been prepared, either in terms of sectors or in terms of screens.

Finally, it should be noted that this file system was generated for a single-density disk system. Accordingly, some changes will be required in word definitions where higher, or mixed, density is involved.

The attached listing (page 42) compiles to 3933 bytes. Somewhat less memory is required if DCOM of screen 89 is not required.

Notes on the Listing

Since the listing is rather sparsely commented, the following is offered as an aid to understanding the file system. In screen 88 a few convenient, but not necessarily essential, constants and variables have been defined. Their main excuse for existence is as a mnemonic aid for rereading the listing six months from now.

Screen 89 defines the only CODE word used in the file system. It links to the North Star DOS routine of the same name in order to perform disk READ/WRITE operations. It is, of course, disk system peculiar. At the time that this system was developed, there was no documented word of similar nature from Omniforth. Since then, Omniforth has circulated a newsletter on RWDSK, a previously undocumented word in Omniforth which behaves in an identical fashion to DCOM. I have left DCOM in the listing in order to illustrate how such a linkage is effected, but am now using RWDSK on my own system. (Why waste memory?)

The input utilities GET-NUM, GET-HEX, and GET-NAME are used to allow error recovery and operator command abort during command prompt input.

The routine ?ENTRY in screen 98 prompts the operator for a file-starting screen, the last screen of the file, and the file type. If the file is type 1, the operator is further prompted for a hexadecimal load-and-go address.

The routine ?EMPTY in screen 99 searches for an empty location in the directory. It expects to find the RAM address of the directory in the variable SAVE. If an empty location is found, the variable BUFR will contain the RAM start address of the empty location and a true flag will be on the stack. Failure to find an empty location will leave a false flag on the stack.

The routine COMPARE in screen 100 compares two strings for equality. It expects ADDR1 ADDR2 LENGTH . . . on the stack. It leaves a false flag on the stack if the strings are not equal, a true flag if equal; now ADDR1 points to the byte following the string associated with the original value of ADDR1; and ADDR2 has been dropped.

The routine FIND-FILE in screen 101 uses COMPARE to locate the file name in IBUF within the directory. On success there is a true flag on the stack and the variable BUFR contains a pointer to the byte following the located name in the directory. On failure, there is a false

flag on the stack. The name was placed in IBUF by GET-NAME.

WRITE-DIREC in screen 101 expects the variable SAVE to contain the pointer to the start of the directory in RAM memory. SAVE was initialized by RD-DIREC as is the drive pointer variable DR. WRITE-DIREC writes the updated directory back to disk. DELETE-ENTRY expects the contents of BUFR to point to the entry to be deleted. The entry is deleted by filling that location with 16 blanks and then writing the updated directory back to disk. DO-ENTRY first checks for an empty location in the directory, and if one is found, moves the contents of IBUF to the location pointed to by the contents of BUFR. The directory is then written back to disk.

NXT-SEC# of screen 104 locates the next free sector address above the "highest" file in the directory. Note that the diskettes are not "squished" when a file is deleted from the directory.

DISK-ALLOT is used by FILE to build a type 5 file entry and put it in the directory.

?SPACE of screen 105 is used to make sure that the file being created or written to a disk will not cause a disk overflow.

It is hoped that these brief notes will help the reader to decipher those routines which have not been annotated on the screens.

■■J

Fig-Forth Directory & File System

(Text begins on page 38)

```
SCR # 88
  0 ( CONSTANTS, VARIABLES, & CONVERSION FACTORS )
  1
  2
  3 48 CONSTANT "0"  57 CONSTANT "9"  65 CONSTANT "A"
  4 70 CONSTANT "F"  90 CONSTANT "Z"  3 CONSTANT CNTRL-C
  5 HEX 2020 CONSTANT BLBL DECIMAL   ( A DBL BLANK )
  6
  7
  8 0 VARIABLE BUFR  0 VARIABLE FLAG  0 VARIABLE SAVE
  9 0 VARIABLE CNTR  0 VARIABLE DR    0 VARIABLE DW
 10 0 VARIABLE #SEC  0 VARIABLE LAST-SEC  0 VARIABLE SECR
 11 0 VARIABLE SECW
 12 ( SCRN2SEC   SCR# ... SEC# )
 13 : SCRN2SEC 87 /MOD DROP 4 * ;
 14 ( SEC2SCRN   SEC# DR# ... SCR# )
 15 : SEC2SCRN 1- 87 * SWAP 4 /MOD SWAP DROP + ;   -->

SCR # 89
  0 ( DCOM - LINKAGE TO DISK READ/WRITE ROUTINE OF NORTH STAR DOS )
  1
  2 ( DCOM - RAM-ADDR SEC-STRT #SEC R/W DR# ....      )
  3 ( R/W=1 FOR READ, =0 FOR WRITE                    )
  4
  5 HEX CODE DCOM
  6    I' L MOV  I H MOV       ( PRESERVE II' )
  7    XTHL  L I' MOV          ( DR# TO C )
  8    H POP XTHL  L I MOV     ( R/W TO B )
  9    H POP XTHL  L A MOV     ( #SEC TO ACC )
 10   H POP XTHL  XCHG        ( DSK ADDR TO DE )
 11   H POP XTHL  XCHG     ( RAM ADDR TO DE, DSK ADDR TO HL )
 12   2022 CALL  I POP ( CALL DCOM & RSTR II' )
 13   0 H MVI  CS IF 1 L MVI  ELSE 0 L MVI ENDIF  ( ERROR FLAG )
 14   H PUSH  NEXT JMP  DECIMAL
 15 -->
```

```
SCR # 90
  0 ( LI & LI.S )
  1 ( THERE ARE TWO DIRECTORY LISTING COMMANDS, LI & LI.S )
  2 ( LI LISTS THE DIRECTORY IN NORTH STAR FORMAT, THAT IS )
  3 ( NAME, SECTOR START, FILE LENGTH IN SECTORS, FILE TYPE )
  4 ( & GO ADDRESS IF APPLICABLE.  THE LI.S COMMAND REPLACES )
  5 ( THE SECTOR INFORMATION WITH SCREEN START & SCREEN STOP )
  6
  7 ( DR-TST -      DR# ... DR#  ABORTS IF DR# OUT OF RANGE )
  8 : DR-ERR CR ." DR# ERROR " CR ABORT ;
  9 : DR-TST DUP 1 < IF DR-ERR ELSE DUP 3 > IF DR-ERR
 10         ENDIF ENDIF ;
 11
 12
 13 : RD-DIREC 0 BUFFER DUP DUP SAVE ! BUFR ! 0 4 1 DR @
 14      DCOM DISK-ERROR ! ;
 15 -->

SCR # 91
  0 ( LI & LI.S CONTINUED )
  1 : ?SCRN FLAG @ IF DR @ SEC2SCRN ENDIF ;
  2 : LIST-DIREC BUFR @ 16 BUFR +! DUP 8 TYPE 8 + DUP @ DUP
  3     2 SPACES ?SCRN 3 .R 2 SPACES SWAP 2+ DUP @ ROT
  4  FLAG @ IF + ?SCRN 1- ELSE DROP ENDIF 3 .R 2 SPACES 2+ DUP C@
  5     DUP 3 .R 1 = IF 2 SPACES 1+ @ .4H ELSE DROP ENDIF ;
  6 : DOUT BUFR @ @ BLBL = IF 16 BUFR +! ELSE LIST-DIREC CR ENDIF ;
  7 : .LI CR DR-TST DR ! FLUSH EMPTY-BUFFERS RD-DIREC CR 0 CNTR !
  8     BEGIN CNTR @ 1 CNTR +! 64 < WHILE DOUT REPEAT ;
  9
 10 ( LI.S -  DR#... LISTS FNAME,SCR# STRT,SCR# STOP,TYPE,GA )
 11 : LI.S 1 FLAG ! .LI ;
 12
 13 ( LI  - DR# ... LISTS FNAME,SEC# STRT,FILE LEN,TYPE,GA )
 14 : LI  0 FLAG ! .LI ;
 15 -->

SCR # 92
  0 ( THE CD COMMAND )
  1 ( CD    DR#1 DR#2 ...    )
  2 ( COPIES ENTIRE DISKETTE ON DR#1 TO DISKETTE ON DR#2 )
  3
  4 : NEXT-SEC #SEC @ DUP SECR +! SECW +! LAST-SEC @ SECR @
  5            #SEC @ + - DUP 0< IF #SEC +! ELSE DROP ENDIF
  6            #SEC @ 0= ;
  7
  8
  9 : DWRITE LIMIT SECW @ #SEC @ 0 DW @ DCOM DISK-ERROR ! ;
 10 : DREAD LIMIT SECR @ #SEC @ 1 DR @ DCOM DISK-ERROR ! ;
 11
 12 ( ABORT COMMAND ON A CNTRL-C )
 13 : ?ABORT CNTRL-C = IF DROP QUIT ELSE DUP ENDIF ;
 14
 15 -->
```

(Listing continued on next page)

Fig-Forth Directory & File System

(Listing continued, text begins on page 38)

```
SCR # 93
  0 ( THE CD COMMAND CONTINUED )
  1 ( THE WORD SET-#SEC FINDS THE AVAILABLE NUMBER OF SECTORS )
  2 ( ABOVE LIMIT THAT CAN BE USED AS A DISK BUFFER.  NORMALLY )
  3 ( THE TOP OF MEMORY IS A CONSTANT, BUT NOT IN MY SYSTEM!! )
  4 : SET-#SEC LIMIT
  5       BEGIN 256 + DUP DUP 0 SWAP ! @
  6 -1 =  UNTIL
  7       256 - LIMIT - 256 / DUP 32 > IF DROP 32 ENDIF #SEC ! ;
  8 ( THE WORD DCOPY IS COMMON TO CD AND CSEC )
  9 : DCOPY  BEGIN DREAD DWRITE NEXT-SEC UNTIL
 10         CR ." COPY COMPLETED " CR ;
 11 : CD SWAP DR-TST DUP DR ! ." COPY FRM DSK " . DR-TST
 12      DUP DW ! ." TO DSK " . CR
 13      ." HIT SPACE BAR TO CONTINUE, ANY OTHER TO ABORT "
 14      KEY 32  IF CR 350 LAST-SEC ! 0 DUP SECR ! SECW !
 15          SET-#SEC DCOPY ELSE ." CMND ABORTED " CR ENDIF ;  -->

SCR # 94
  0 ( THE CSEC COMMAND )
  1 ( CSEC    SEC#1  SEC#2  DR#1  SEC#3  DR#2  ...       )
  2 ( COPIES SEC#1 THRU SEC#2 FROM DRIVE DR#1 TO DR#2  )
  3 ( STARTING AT SEC#3. )
  4
  5 : CSEC DR-TST DW ! SECW ! DR-TST DR ! 1+
  6        LAST-SEC ! SECR ! SET-#SEC LAST-SEC @ SECR @
  7      -  DUP #SEC @ < IF #SEC ! ELSE DROP ENDIF DCOPY ;
  8
  9 ( THE FOLLOWING SEVERAL SCREENS DEVELOP A SET OF UTILITIES. )
 10 ( THE WORD BUFF IS USED TO CREATE TEMPORARY HOLDING BUFFERS )
 11 ( FOR COMMAND PARAMETER INPUT. )
 12
 13 ( BUFF -    N1 BUFF NAME       ALLOTS N1 BYTES TO NAME )
 14 : BUFF <BUILDS ALLOT DOES> ;
 15 16 BUFF IBUF   16 BUFF TBUF   -->

SCR # 95
  0 ( GET-NUM  LEAVES DECIMAL # ON TOP OF STACK )
  1 : SYN-ERR CR ." SYNTAX ERROR " DROP DROP 0 "0" CR ;
  2 : GET-NUM  0 FLAG ! ( QUIT WHEN FLAG = 1 )
  3    0 BEGIN  ( PUT INITIAL # = 0 ON TOP OF STACK )
  4           KEY DUP DUP EMIT ?ABORT ( GET CHR & ECHO IT )
  5       13 = IF 1 FLAG ! DROP  ( QUIT ON A CR )
  6          ELSE DUP "0" < IF  ( CHR < 0 ? )
  7                         SYN-ERR
  8          ELSE DUP "9" > IF  ( CHR > 9 ? )
  9                         SYN-ERR ENDIF ENDIF ENDIF
 10 ( NOW CNVT ASCII TO DECIMAL & ADD TO THE STACK )
 11  FLAG @ 0= IF 48 - SWAP 10 * + ENDIF
 12
 13   FLAG @ UNTIL ; ( KEEP GOING TILL CR )
 14
 15 -->
```

```
SCR # 96
  0 ( GET-HEX  LEAVE A HEX # ON TOP OF STACK )
  1
  2
  3
  4 : GET-HEX 0 FLAG ! 0 BEGIN KEY DUP DUP EMIT ?ABORT
  5           13 = IF 1 FLAG ! DROP
  6              ELSE DUP "0" < IF SYN-ERR
  7                ELSE DUP "9" > IF DUP "A" < IF SYN-ERR
  8              ELSE DUP "F" > IF SYN-ERR ENDIF
  9              ENDIF ENDIF ENDIF ENDIF
 10  FLAG @ 0= IF 48 - DUP 9 > IF 7 - ENDIF SWAP 16 * + ENDIF
 11
 12    FLAG @ UNTIL ;
 13 -->
 14
 15


SCR # 97
  0 ( GET-NAME )
  1 : INIT-IBUF IBUF DUP BUFR ! 8 CNTR ! 0 FLAG ! IBUF 16 BLANKS ;
  2 : NM-ERR CR ." SYNTAX ERROR " CR ." REENTER FNAME  "
  3          DROP DROP INIT-IBUF ;
  4 : GET-NAME INIT-IBUF  ." FNAME? "
  5         BEGIN KEY DUP DUP EMIT ?ABORT
  6            13 = IF 1 FLAG ! DROP
  7                 ELSE DUP 33 < IF NM-ERR
  8                              ELSE DUP "Z" > IF NM-ERR
  9                                             ELSE BUFR @ 1 BUFR
 10                                              +! C! -1 CNTR +!
 11                                             ENDIF
 12                              ENDIF
 13                 ENDIF CNTR @ 0= IF 1 FLAG ! ENDIF
 14   FLAG @ UNTIL DROP CR ;
 15 -->


SCR # 98
  0 ( ?ENTRY AND CLR-SCR )
  1
  2 : ?ENTRY  ." SCR STRT?  " GET-NUM CR DUP SCRN2SEC
  3           IBUF 8 + DUP 2+ BUFR ! !
  4           ." LAST SCR?  " GET-NUM CR SWAP - 1+ 4 *
  5           BUFR @ ! 2 BUFR +!
  6           ." FILE TYPE?  " GET-NUM CR DUP DUP 5 = IF
  7            ." WRONG FILE TYPE " CR ABORT ELSE BUFR @ C! 1
  8         BUFR +! ENDIF
  9    1 = IF ." HEX GO ADDR?  " GET-HEX CR BUFR @ ! ENDIF ;
 10
 11 ( CLR-SCR  -    SCR#1 SCR#2 .... CLEARS SCR#1 THRU SCR#2 )
 12
 13 : CLR-SCR 1+ SWAP DO I EDITOR CLEAR FORTH LOOP ;
 14
 15 -->
```

(Listing continued on page 46)

Fig-Forth Directory & File System

(Listing continued, text begins on page 38)

```
SCR # 99
  0 ( ?EMPTY )
  1 : ?EMPTY 0 DUP CNTR ! FLAG ! SAVE @ BUFR !
  2          BEGIN 1 CNTR +! BUFR @ 16 BUFR +!
  3                @ BLBL = IF -16 BUFR +! 1 FLAG !
  4                        ELSE CNTR @ 65 = IF
  5                        1 FLAG ! ENDIF ENDIF
  6  FLAG @ UNTIL
  7          CNTR @ 65 = IF 0 ELSE 1 ENDIF ;
  8
  9
 10 -->

SCR # 100
  0 ( COMPARE )
  1 : COMPARE 1 FLAG !
  2           0 DO DUP C@ ROT DUP C@ ROT
  3       = IF 1+ SWAP 1+ ELSE 0 FLAG !  ENDIF LOOP
  4       SWAP DROP FLAG @ DUP 0= IF SWAP DROP ENDIF ;  -->
  5 ( COMPARE TWO STRINGS ADDR1 ADDR2 CNT ON STACK,
  6   LEAVE 0 ON STACK IF NOT EQUAL, 1 IF EQUAL & ADDR1 PNTS
  7  TO CELL FOLLOWING NAME )
  8
  9
 10 -->

SCR # 101
  0 ( FIND-FILE WRITE-DIREC DELETE-ENTRY DO-ENTRY )
  1 : FIND-FILE 0 CNTR ! SAVE @ BUFR ! 0 FLAG !
  2         BEGIN 1 CNTR +! IBUF BUFR @ 16 BUFR +! 8
  3               COMPARE
  4               IF DROP -16 BUFR +! 1 FLAG ! ENDIF
  5               CNTR @ 65 = IF 1 FLAG ! ENDIF
  6  FLAG @ UNTIL
  7  CNTR @ 65 = IF 0 ELSE 1 ENDIF ;
  8
  9 : WRITE-DIREC SAVE @ 0 4 0 DR @ DCOM DISK-ERROR ! ;
 10
 11 : DELETE-ENTRY BUFR @ 16 BLANKS WRITE-DIREC ;
 12
 13 : DO-ENTRY ?EMPTY IF IBUF BUFR @ 16 CMOVE WRITE-DIREC
 14                   ELSE CR ." DIRECTORY IS FULL " CR QUIT
 15                   ENDIF ;    -->

SCR # 102
  0 ( DE )
  1 ( DE -     DR# ...  PROMPTS FOR FNAME )
  2 ( USED TO DELETE NAMED FILES FROM THE DIRECTORY OF THE )
  3 ( SPECIFIED DRIVE DR# )
  4
  5 : CHK-NAME DR-TST DR ! RD-DIREC GET-NAME CR FIND-FILE ;
  6 : DE CHK-NAME IF DELETE-ENTRY ELSE ." FILE NOT FOUND "
  7      CR ENDIF ;
  8 -->
```

```
SCR # 103
  0 ( LIST-FILE )
  1 ( LIST-FILE   -    DR# ... PROMPTS FOR FILE NAME  )
  2 ( USED TO LIST NAMED FILE FROM THE SPECIFIED DRIVE )
  3 ( WILL ACCEPT FORTH SOURCE PROGRAMS OF TYPE 4 ONLY )
  4
  5 ( : .FF 12 EMIT ; )
  6 : LIST-FILE CHK-NAME IF BUFR @ DUP 12 + C@ 4 = IF .FF 8 +
  7              DUP @ 4 / DR @ 1- 87 * + SWAP 2+ @ 4 /
  8            0 CNTR ! 0 DO CNTR @ 3 = IF .FF
  9            0 CNTR ! ENDIF DUP LIST 1+ 1 CNTR +! LOOP
 10          DROP .FF ELSE DROP ." WRONG FILE TYPE " CR
 11          QUIT ENDIF ELSE CR ." FILE NOT FOUND " CR ENDIF ;
 12 -->

SCR # 104
  0 ( NXT-SEC# )
  1 : NXT-SEC# RD-DIREC 0 DUP DUP CNTR !
  2            BEGIN 1 CNTR +! BUFR @ 16 BUFR +!
  3                  DUP @ BLBL = IF DROP
  4                  ELSE 8 + DUP @ SWAP 2+ @ + DUP ROT
  5                  > IF SWAP DROP DUP ELSE DROP DUP ENDIF
  6                  ENDIF CNTR @ 65 = UNTIL DROP ;
  7 : DISK-ALLOT IBUF 16 BLANKS IBUF 8 + !
  8     4 * IBUF 5 OVER 12 + C! 10 + !
  9     DUP 1+ SWAP C@ 31 AND
 10    IBUF SWAP DUP >R CMOVE R> IBUF + 1- DUP C@ 127 AND SWAP C!
 11       FIND-FILE IF CR ." FILE NAME CONFLICT " CR QUIT ELSE
 12     DO-ENTRY ENDIF ;
 13
 14
 15 -->


SCR # 105
  0 ( FILE )
  1 ( #SCRNS DR#   FILE   FNAME )
  2 ( BYTE# DR#   FNAME      LEAVES RAM ADDR OF BYTE# )
  3
  4 ( ?SPACE     #SCRNS SEC-STRT ... #SCRNS SEC-STRT 1 )
  5 (              #SCRNS SEC-STRT ... 0                )
  6 : ?SPACE 2DUP SWAP 4 * + 350 > IF DROP DROP DROP 0
  7                              ELSE 1 ENDIF ;
  8
  9 : FILE DR-TST DR ! HERE SWAP NXT-SEC# ?SPACE
 10   0= IF ." INSUFFICIENT DISK SPACE " CR QUIT ENDIF
 11      <BUILDS  DUP , DISK-ALLOT
 12      DOES> @ SWAP DUP DR ! 1- 174 * SWAP 2 / +
 13            SWAP 512 /MOD ROT + BLOCK + UPDATE ;
 14
 15 -->


SCR # 106
  0 ( ND )
  1 ( ND   -    DR# ... PROMPTS FOR DISK NAME  )
  2 ( PROVIDES A MEANS OF UNIQUELY IDENTIFYING EACH DISK )
```

(Listing continued on next page)

Fig-Forth Directory & File System

(Listing continued, text begins on page 38)

```
 3 ( AS THE FIRST ENTRY IN THE DIRECTORY.  IT RESERVES )
 4 ( THE FIRST 4 SECTORS OF THE DISK FOR THE DIRECTORY. )
 5 ( THIS COMMAND WILL OVERWRITE ANY PRE-EXISTING FIRST ENTRY. )
 6
 7
 8 : ND DR-TST DR ! RD-DIREC GET-NAME IBUF SAVE @ 8 CMOVE
 9       SAVE @ 8 + DUP @ SWAP ! 2+ DUP 4 SWAP !
10       2+ DUP @ SWAP C! 1+ @ SWAP ! WRITE-DIREC ;
11
12
13
14
15 -->


SCR # 107
  0 ( CSCR )
  1 ( CSCR  -  SCR#1 SCR#2 SCR#3 ...                    )
  2 ( COPIES FROM SCR#1 THRU SCR#2 TO SCR#3 & UP        )
  3
  4
  5
  6
  7
  8
  9 : CSCR SECW ! 1+ SWAP DO I SECW @ 1
 10      SECW +! EDITOR COPY FORTH LOOP ;
 11 -->

SCR # 108
  0 ( CRF )
  1 ( CRF  -    DR# ...  PROMPTS FOR OTHER PARAMETERS    )
  2 ( USED TO CREATE DIRECTORY ENTRIES FOR ANY FILE TYPE )
  3 ( EXCEPT TYPE 5, THAT FORTH DATA FILES ARE INHIBITED. )
  4
  5
  6 : CRF DR-TST DR ! RD-DIREC GET-NAME ?ENTRY FIND-FILE
  7       IF CR ." FILE ALREADY EXISTS " CR
  8       ELSE DO-ENTRY ENDIF ;
  9 -->

SCR # 109
  0 ( LOAD-FILE )
  1 ( LOAD-FILE  -   DR# .... PROMPTS FOR FNAME         )
  2 ( USED TO LOAD FORTH SOURCE PROGRAM FILES.           )
  3 ( ACCEPTS TYPE 4 FILES ONLY. )
  4
  5
  6
  7 : LOAD-FILE CHK-NAME
  8           IF BUFR @ 8 + @ DR @ SEC2SCRN LOAD
  9             ELSE CR ." FILE NOT FOUND " CR ENDIF ;
 10 -->
```

```
SCR # 110
  0 ( CF   )
  1 ( CF   -    DR#1 DR#2 ... PROMPTS FOR FILE NAMES     )
  2 ( COPIES SPECIFIED FILE FROM DRIVE DR#1 TO SPECIFIED )
  3 ( FILE ON DRIVE DR#2.  FILE NEED NOT EXIST ON DR#2,  )
  4 ( SINCE ROUTINE WILL CREATE THE NECESSARY ENTRY IF )
  5 ( THERE IS ROOM ON THE DISK.  IF THE FILE ALREADY  )
  6 ( EXISTS ON DR#2 THEN THE COPY PROCEEDS ONLY IF THE )
  7 ( FILE IS SUFFICIENTLY LARGE. THAT IS >= )
  8
  9 0 VARIABLE DT   0 VARIABLE DF        16 BUFF FBUF
 10 : WRITE-SEC FBUF 8 + DUP @ ( START SEC )
 11         DUP ROT 2+ @ + 1-        ( LAST SEC )
 12         DF @ ( FROM DRIVE )
 13         TBUF 8 + @ ( FIRST TO-SECTOR )
 14         DT @ ( TO-DRIVE )
 15         CSEC ( XFR SECTORS ) ; -->

SCR # 111
  0 ( CF CONTINUED )
  1 : CF SWAP DUP ." FROM DRIVE " . ." & FROM " CHK-NAME
  2 IF DR @ DF ! BUFR @  FBUF 16 CMOVE ( ENTRY TO FBUF, DF )
  3    ELSE ." FILE NOT FOUND " CR QUIT ENDIF
  4  DUP ." TO DRIVE " . ."  & TO  " CHK-NAME
  5  IF DR @ DT ! BUFR @  TBUF 16 CMOVE ( ENTRY TO TBUF,DT )
  6       TBUF 10 + @ FBUF 10 + @ 1- - ( LEN DIFF => 0 IS OK )
  7       IF WRITE-SEC QUIT ENDIF
  8    ELSE ( CREATE A NEW ENTRY ON TO-DRIVE )
  9       DT @ DW ! FBUF IBUF 16 CMOVE NXT-SEC# DUP IBUF 8 + !
 10      IBUF 10 + @ + DUP 350 > IF ." INSUFFICIENT DISK ROOM "
 11                                 CR QUIT
 12                               ELSE DO-ENTRY ENDIF
 13        ( NOW WRITE THE SECTORS TO THE DRIVE )
 14      IBUF TBUF 16 CMOVE WRITE-SEC
 15  ENDIF ;
```

End Listing

SAY" Forth Votrax Driver"

by C. Kevin McCabe

Forth enthusiasts frequently cite modularity and expandability as the language's greatest strengths. The truth of these claims can be shown with an example: Design of all necessary software to interface a Votrax Type'N Talk speech synthesizer to a system.

BASIC and other "conventional" languages make such software interfaces difficult to design and inconvenient to use. In general, all or most of the interface routine must be written in machine language; portability between systems, even within the same processor family, is rare. The routine must be explicitly included in every program requiring use of the peripheral, by either keying in a subroutine or loading a utility module from mass storage. Commands within the BASIC program (let alone the machine code routines) are generally not self-documenting, with obscure references like K=USR(7) or X=INP(231) AND &127 perhaps meaning to initialize the proper port or check an input bit.

With Forth, though, the entire software interface can be programmed in portable high-level code, with virtually no penalty to execution speed and memory requirements. The resulting interface is hardly an inelegant "patch job," as in BASIC; after development, the interface becomes part of the language, usable in exactly the same manner as any other Forth "words." It is even possible to make the interface software part of the core language, so that it will be available immediately upon a cold start without an additional loading operation. Perhaps best of all, interactive development and testing make the software design process incredibly fast and efficient – in this instance, the Votrax software was developed from first idea to final test in under 20 minutes, and still uses less than 100H bytes of memory.

The basic function of such an interface routine is to move data from memory to the peripheral, in accordance with the data conventions of both the language and the device. The Votrax Type'N Talk is designed to receive ASCII data via an RS-232C port, using any of eight baud rates. At low data speeds, and with short and relatively infrequent output strings, the physical interface does not need handshaking lines. Longer, faster, or more frequent output, though, will overflow the Votrax's limited buffer unless handshaking is implemented using the Type'N Talk's Clear To Send line. In this instance, Clear To Send is simply brought into a single control port bit on a Vector Graphics serial I/O board; other systems may use different handshaking schemes appropriate to the hardware.

C. Kevin McCabe, 115 South LaSalle, Suite 1200, Chicago, IL 60603.

The interface routine in Listing 1 uses the Forth Interest Group's Forth (Fig-Forth) for the 8080, release 1.1. With minor changes at most, the software should be portable to other Fig-Forth installations regardless of processor types. Only CLEARTHROAT and two or three integer constants in SAYCHARACTER are system-dependent. In this case, CLEARTHROAT is defined to output the six data bytes shown in line 4 of screen 500 to serial control port 07H. These bytes initialize serial ports 06H and 07H of the I/O board for the appropriate word size and baud rate. If the system uses serial ports which are integral to the system or initialized upon cold start, CLEARTHROAT or its equivalent may not be necessary. In any event, CLEARTHROAT is only executed once, during the loading operation.

The core Fig-Forth language, available after a cold start, contains over a dozen words for terminal output. Like most of the language, the "building block" approach has been used extensively in these word definitions. In most instances, only EMIT (which takes an ASCII character byte from the stack and sends it to the terminal) is defined as a machine-language "primitive;" all other terminal output builds upon EMIT to output strings and numerics.

The same scheme can be used for output to the Votrax. As defined in lines 11-14 of screen 500, SAYCHARACTER treats the value currently on top of the stack as an ASCII character byte, and outputs it to the Votrax (and optionally to the terminal as well) with full handshaking. No machine code is required, as the core Fig-Forth language already provides all necessary operations in high level.

SAYCHARACTER executes by first entering an indefinite BEGIN...UNTIL loop to read input from port 07H, repeating until a Clear To Send signal is received. For the hardware used here, bit 0 on port 07H is the handshaking signal; a zero on that bit signals that the Votrax's input buffer is full. The loop will terminate whenever bit 0 goes to a 1. Line 13 duplicates the top stack value for later use, then outputs the copy to the Votrax via serial port 06H.

For convenience, one additional feature is added to the definition of SAYCHARACTER. At times, it may be necessary to send output to both the Votrax and the terminal. The most efficient means is to include a software switch in the lowest-level output definition; in that way, all later words defined with this "building block" will use the same method to direct output, at no additional software cost. Here, SAYCHARACTER fetches the value of the variable VOICE+PRINT for use as a logical flag. A non-zero value (a true flag) sends the remaining copy of the character byte to the terminal, using EMIT for output. If the flag value is zero (false), the remaining copy is simply dropped from the stack.

Screens 501 and 502 contain the remaining high-level words to incorporate ASCII string audio output capabilities to the Forth system; all are system-independent, as they are defined using SAYCHARACTER. For ease of use, the words are defined similarly to the Fig-Forth core words for terminal output, with the same stack values and formats.

SPEAK, like TYPE for terminal output of strings, requires two operands from the top of the stack. The top value *n* is the character count of the string whose first data byte is at address *addr*. Any non-zero count will fetch *n* ASCII characters one-by-one from memory to the stack, and then send each to the Votrax. If the count byte is zero, SPEAK simply removes the address operand without output of any characters. Since output of individual characters uses SAYCHARACTER, each may also be directed to the terminal by first storing a true flag value in the VOICE+PRINT variable. Note that any character string in memory can be output using SPEAK, so long as the initial data byte address and character count are first placed on the stack.

SAY" and (SAY") operate in tandem during various phases of operation, in the same manner as the terminal output word ." and its (.") run-time procedure. Character strings delimited by a trailing "quote" character (22H = 34 decimal) are input from the terminal, separated from SAY" by one or more blanks. Further execution of SAY" depends upon Forth's

current operating state; a zero value for the user variable STATE indicates that Forth is executing rather than compiling.

In the execution state, lines 10-12 of screen 502 move the input string from the terminal buffer to the top of the dictionary, with a leading one-byte character count. The current dictionary pointer is placed on the stack, then used by COUNT to increment the pointer and fetch the contents of the count byte. These two values are the operands for SPEAK, which output the string to the Votrax. Again, VOICE+PRINT can be used to direct output to the terminal as well.

When SAY" is encountered as a component word during compilation of colon definition – that is, in Forth's compilation state – the action is somewhat more complex. SAY" is defined as an immediate word, which executes even during the compilation state. It does not, though, cause immediate output as during execution. Instead SAY" first compiles an address pointer to the (SAY") run-time routine. The text string, along with its leading count byte, is moved to the next available dictionary locations. Finally, the dictionary pointer is incremented to the byte beyond the end of the string.

During later execution of the colon-defined word, execution of (SAY") will output the compiled string. Fig-Forth uses the return stack to hold an address pointer to the next component of a colon-defined word; in this instance, the next "word" is actually the string compiled into the definition by SAY". Lines 11 and 12 of screen 501 non-destructively copy the pointer to the string from the return stack and increment it to designate the first data byte. That pointer is left on the stack, along with two copies of the character-count byte. One copy of the count is incremented (to account for the count byte itself), and used in turn to increment the return stack's word pointer to beyond the string. Finally, SPEAK is used to output the string.

That's it – all the basic building blocks needed for audio output of ASCII strings with the Votrax Type'N Talk. Listing 2 defines several other words using these fundamental components, for convenient output of numeric values and ASCII text from disk screens. Again, all are completely portable, and all use VOICE+PRINT as a software switch – the direct result of Forth's building block approach.

SAYLINE corresponds to Fig-Forth's .LINE and outputs line *n1* of screen *n2* to the Votrax. Similarly, SAYSCREEN outputs an entire screen – presumably ASCII text, since Forth source code's abbreviations and symbols don't translate well to audio.

Single- and double-precision numeric stack values may be output to the audio system with SAYU, SAYN, and SAYD, respectively, in the same manner as U., ., or D. for printed output. If numeric output to the terminal is desired as well, it is easiest to use separate voice and terminal output words in this instance rather than the VOICE+PRINT software switch. As defined, SAYD outputs a negative sign as the word "negative" rather than a single character. In addition, digit characters are output separated by blanks; otherwise, the Votrax would attempt to pronounce a value such as 0CABH as a single word rather than the desired "see-a-bee."

Aside from any necessary port initialization, only SAYCHARACTER will require modification for use on other systems. Appropriate values should be used for the control port number in line 12 and data port number in line 13 of screen 500. If the Votrax's Clear To Send control signal is not brought onto the I/O board as bit 0, it will also be necessary to change the masking constant preceding the logical AND operation in line 12.

Users of the proposed Forth-79 version of the language should note that more extensive changes will be necessary. Some Forth-79 words are defined somewhat differently than their Fig-Forth counterparts; in other instances, no Forth-79 counterpart exists. Examples of the latter problem are the words P! and P@, which transfer data between the stack and I/O ports. Fortunately, most Forth-79 implementations are also supplemented with these and other useful Fig-Forth words.

The Forth-79 version of WORD moves text strings in much the same manner as the Fig-Forth version, but also leaves a pointer to the leading count byte on the stack; remove HERE from both locations in the definition of SAY" and use the address left by WORD instead. Also, the initial value of VOICE+PRINT should not precede the definition, as Forth-79 variables must be initialized after being defined. Finally, Forth-79 does not explicitly require the same type of return stack usage as Fig-Forth (although as a practical matter, few Forth-79 systems will differ). Some experimentation may be necessary to determine if (SAY") will operate in the proper manner.

DDJ

(Listing begins on page 58)

SAY''

(Text begins on page 55)

Listing One

```
Screen # 500
  0 ( Type'N Talk Output-1                          CKM 06 Oct 82      )
  1 HEX
  2
  3 : CLEARTHROAT ( initialize Vector Graphics I/O port)
  4     27 0CE 40 0 0 0 ( data bytes)
  5     6 0 DO 7 P! LOOP ( output 6 bytes to port 7) ; DECIMAL
  6
  7 CLEARTHROAT ( execute to initialize port)
  8
  9 0 VARIABLE VOICE+PRINT ( 0=voice only, 1=say and print too)
 10
 11 : SAYCHARACTER ( c --> )
 12     BEGIN 7 P@ 1 AND UNTIL ( wait for clear to send)
 13     DUP 6 P! ( say the character)
 14     VOICE+PRINT @ IF EMIT ELSE DROP THEN ( print if desired) ;
 15 -->
Screen # 501
  0 ( Type'N Talk Output-2                          CKM 06 Oct 82      )
  1
  2 : SPEAK ( addr n >> )  ( based on TYPE)
  3     -DUP ( duplicate non-zero count byte)
  4     IF ( valid count) OVER + SWAP ( find loop parameters)
  5         DO I C@ ( get next character)
  6         SAYCHARACTER ( output character) LOOP
  7     ELSE ( zero count) DROP
  8     THEN ;
  9
 10 : (SAY")  ( run-time routine based on (.")
 11     R ( copy pointer to next word) COUNT DUP ( get byte count)
 12     1+ R> + >R ( increment pointer to next word)
 13     SPEAK ( output string) ;
 14
 15 -->
Screen # 502
  0 ( Type'N Talk Output-3                          CKM 06 Oct 82      )
  1
  2 DECIMAL
  3
  4 : SAY" ( immediate word based on ." )
  5     34 ( Ascii of " delimiter)
  6     STATE @ ( fetch compilation state flag)
  7     IF ( compiling) COMPILE (SAY") ( compile run-time pointer)
  8         WORD ( move string to top of dictionary)
  9         HERE C@ ( fetch count byte) 1+ ALLOT ( increment dp)
 10     ELSE ( executing) WORD ( move string to top of dictionary)
 11         HERE COUNT ( leave address & count)
 12         SPEAK ( output string)
 13     THEN ;
 14 IMMEDIATE
 15 ;S
```

Listing Two

```
Screen # 503
  0 ( Type'N Talk screen utilities                  CKM 10 Oct 82      )
  1 DECIMAL
  2
  3 : SAYLINE ( n1 n2 >> )  ( based on .LINE)
  4     (LINE) ( move screen n2 to buffer, leave starting addr and)
  5            ( character count of line)
  6     -TRAILING ( adjust count for trailing blanks)
  7     SPEAK ( output the line) ;
  8
```

```
 9 : SAYSCREEN ( n >> ) ( output Ascii text of entire screen n)
10     SCR ! ( save screen number in user variable)
11     16 0 DO ( set up line count loop)
12        I SCR @ ( leave line, screen numbers)
13        SAYLINE ( say the line)
14     LOOP ;
15 ;S


Screen # 504
 0 ( Type'N Talk numeric utilities              CKM 10 Oct 82      )
 1 DECIMAL
 2 : SAYD ( d >> ) ( based on D.)
 3     DUP ( copy high bytes)
 4     0< IF ( d < 0 ) SAY" NEGATIVE" THEN ( output sign)
 5     DABS <# #S #> ( convert to Ascii, leave start addr and n)
 6     OVER + 1+ SWAP ( get limits for char. by char. output loop)
 7     DO 32 SAYCHARACTER I C@ SAYCHARACTER 32 SAYCHARACTER
 8     LOOP ( output with blanks between characters) ;
 9
10 : SAYN ( n >> ) ( based on . )
11     S->D ( convert n to equivalent d) SAYD ( output d) ;
12
13 : SAYU ( u >> ) ( based on U.)
14     0 ( leave a dummy high-order half of d) SAYD ( output d) ;
15 ;S
```

End Listings

A TRS-80 8080 to Z80 Translator

Anthony T. Scarpelli

Anthony T. Scarpelli, 98 Foxcroft Drive, Scarborough, ME 04074.

This program is based on Robert W. Dea's original 8080 to Z80 translator, published in DDJ *No. 45. The parts of his original work that appear here are used with his permission. Mr. Dea has now placed his original translator in the public domain. Mr. Scarpelli's work here is also in the public domain.*

Back in the May 1981 issue of *DDJ*, No. 45, Robert W. Dea offered us his program "An 8080 to Z80 Translator System." Following is a description of the program and the many modifications I have made to it that will allow the creation of an 8080 source file, and through translation, will produce a Z80 source file that will be accepted by Radio Shack's, or any equivalent, editor/assembler.

A rather large project I had been considering for some time was to convert the Forth Interest Group's 8080 version of Forth into Z80 code. Since I had never bothered to learn the 8080 mnemonics, and was well versed in the Z80 codes, I thought a translated version would afterward allow easier debugging and modification of the Forth program. Since my system is a TRS-80, 48K, model I, with a single disk, I had to modify Mr. Dea's translation program to work with Microsoft's BASIC.

The program was also modified extensively to speed up the translation process. By changing his READ statements into a Data Array, and by using a binary search, as well as using a double-speed modification on my system, and also compressing the BASIC program to get rid of spaces and REM statements, I was able to substantially reduce the time it took to convert the 8080 code to Z80 code. For instance, using a test file of 8080 code from POP H to XTHL, I was able to change the translation time from 7 minutes 39 seconds to 3 minutes 18 seconds.

There are probably many other modifications that could be done to increase the speed even further, such as changing multiple lines to single lines, watching how all the variables are dimensioned, streamlining loops, etc., but the time to do this didn't justify the few seconds more I would get. So I left it the way it is now, and it works fine.

Many parts of the program are just as Mr. Dea wrote it, but I added comments in some lines to help me understand what's going on a little better, and I also wrote a flow chart for it. I found my first round with the program tough going because of the jumps around the BASIC code, but now my understanding is a little better, and I have included the flow chart (starting on page 63) so that you can more easily modify the program if you need to.

Let me go through the program somewhat to show you what I did, and to indicate how an editor/assembler file is created. I should start by describing the part of the program that will create the 8080 code file, so jump to line 4810.

Creating the 8080 File

I use Ultrados for my operating system, so the first thing we have to do is make sure a buffer is available; that's the CMD"O" in line 4830. With TRSDOS you don't have need of this code, and it can be deleted. I don't know what has to be done with other DOSs.

The instructions are pretty self-explanatory; the only important thing I should note is that this routine does no editing. The only chance to correct mistakes is when you are asked "SURE?". You are given a chance to change your mind by inputting an "N." When you are rolling along, and enter a lot of code, sometimes you forget to correct mistakes. Generally this causes no harm; the translation will just ignore an opcode that can't be found, or something weird will be added to the translation. This can be normally edited out when the file is in the editor/assembler.

The other important point I must mention is due to the way random access files are written to disk. Only 255 characters are allowed in one record, but there is room for 256. The 256th character is a random character of some type in the Z80 EDTASM source file that can be edited out during the assembly process. It could also be edited out from disk with a zapper program of some type, but usually this is not necessary. When using the EDTASM in the assembly mode, the assembly process can be switched to stop on any errors. This is where all editing of the file can be done. Some day I'll see if this can be fixed. As you can see there is still room for some improvements to this program.

Line 5020 adds an up-arrow (CHR$ (91)) delimiter to the file's end marker. All entries have this delimiter. Lines 5030 and 5040 open the random access file, and set up A$ as the buffer.

When we are adding 8080 code to the file, the only difference between a line label or comment line, and an operation code is whether there is a space in the first column or not. Line 5070 prints

```
--> ; MEMORY TEST
SURE?
    \
-->  MVI C,01 ;LOAD REPEAT COUNTER
SURE?
    \
--> START LXI SP,0FFFFH ;LOCATE STACK COUNTER
SURE?
    \
--> LXI
SURE? N
    \
-->  LXI H,0000H ;LOAD START ADDRESS
SURE?
    \
-->
```

Figure 1.
Output from Create routine.

```
ENTER 8080 FILE NAME ? MEMTST80/TMP
ENTER Z80 DISK FILESPEC ? MEMTSTZ/SOR
ENTER EDTASM SOURCE FILE NAME (6 CHAR. MAX) ? MEMTST
START LINE NUMBERS AT ? 100

- 00100  - ; MEMORY TEST
00100 ; MEMORY TEST
- 00110  -  MVI C,01 ;LOAD REPEAT COUNTER
00110          LD      C,01          ;LOAD REPEAT COUNTER
- 00120  - START LXI SP,0FFFFH ;LOCATE STACK
00120 START    LD      SP,0FFFFH     ;LOCATE STACK
- 00130  -  LXI H,0000H ;LOAD START ADDRESS
00130          LD      HL,0000H      ;LOAD START ADDRESS
- 00140  -  LXI D,1FFFH ;LOAD TOTAL BYTES
00140          LD      DE,1FFFH      ;LOAD TOTAL BYTES
- 00150  -
```

Figure 2.
Output from Translate program.

a down-arrow to mark this column, and line 5080 is where we prompt for an input. See Figure 1 for an example of this process. Note that a down-arrow prints out as a backslash on my printer.

If you were to exceed the 255 character input to a record, you would jump out of the program and get an error statement. To avoid this, lines 5090 and 5100 make sure that you are aware of this and give you a chance to reduce your line somewhat to fit. The record will be written to disk as soon as it exceeds 215 characters and is less than 255. This wastes a few bytes, but is simple to implement.

The rest of the routine takes the input A1$, adds an up-arrow, and adds it to A2$ if the length is right. In line 5200 spaces are padded to A2$ and it is placed into buffer A$. Then it is put onto disk in line 5210. The record number, R, is incremented and we can now jump back to add more 8080 code to the file. An "@EXIT" closes the file.

Now we have a file filled with 8080 code and comments separated by an up-arrow. We take this file and filter it through the translator in the next section.

Translating

Figure 2 (page 60) shows output from the "translator" part of the program. In essence, what the program does is to first get the 8080 instruction, search through an array until it is found in the data, and then create a string of the Z80 code. Next, the 8080 operand is converted to equivalent Z80 code as specified by the subfields in the rest of the data and added to the string. A data line is specified in lines 170-250.

After the menu, we clear string space and dimension the array in line 430. The only reason for line 440 is to make sure the speed goes back to normal if an error occurs. We have to have normal speed during disk accesses. This error trap can be expanded if necessary.

Next we make sure two buffers are available in 460, then comes the array (83 elements long), then some more initialization until line 1410. Here we ask for the same filespec used in the creation of the 8080 source code file. Then we get the filespec used for the Z80 file, and then the source file name, which is required to be only six characters long.

The format of an editor/assembler file is as follows: the first character is a D3, followed by six ASCII characters for the file name. If the name is less than six bytes, the rest is filled up with spaces to make six. Next is the line number, five bytes long, with each decimal number increased by 176 decimal; for example, a 5 becomes a B5H token. Then comes a space, and then the text with a terminator of an 0D. The next line number follows immediately, and so on. The final byte of the file is a 1A.

Lines 1540 to 1590 fill in the name with spaces. The next routine, lines 1620 to 1760, produces two numbers in the form of strings; one is C2$, which is used for printing, and the other, C3$, is for the Z80 output file.

The next routine, lines 1790 to 1880, goes to the disk and gets the next record, line 1810, and then pulls out the 8080 line of code. Remember it is delimited by an up-arrow, CHR$(91). It leaves this routine with the code in I$, which is printed on the screen.

The next few lines, 1910 to 1940, check to see if we are at the end of the 8080 codes. Also, in order to be able to merge the source code in two or more editor/assembler files, it's a good idea to note the last line number so that you can start the next 8080 file with a larger line number.

We have two output buffers, one for printing, 02$, and one that will go to disk, 0$. The routine in lines 1980 to 2000 first checks to see if the first column is a semi-colon. If so, it will just add the text to the buffer. If not, then we go to the subroutine at 4030. Since this is an important routine, let's go to it.

The 8080 code line is delimited by blanks. If the first column is a blank then we know that the line contains no label, so we test for this in line 4040. If there is a character we read through I$ until a blank is encountered. The label, up to the next blank, is edited into the output buffers from 4060 to 4150. Then we add eight spaces so the printout of 02$ looks like it would in an editor/assembler, and then add a tab to the file buffer to delimit it. If there was no label, then we merely skip the editing part of the routine. Next we jump to a common routine that scans the code line (I$) to get the next code out of it. Let's jump to it for a minute. GOSUB 3470.

The lines from 3520 to 3590 just jump over any blanks in the code line. Then lines 3640 to 3760 accumulate the actual code and leave it in N$. Now return to 4240, and return to 2040. Then GOSUB 4270. A lot of this code is original and does a lot of jumping around as you can see. It's not my style, but I didn't want to spend a lot of time turning it into some structured code.

Lines 4270 to 4350 are the binary search routine I added. It's a typical binary search so I won't go into it. The routine exits pointing to the data from the array ready to be worked on. That is, F is the index into the array DA$. Now we can return to 2110, if there was indeed a match.

The first thing we do is to make sure our output buffers are empty. Then we go back to our subroutine at 4030 that merely looks at the beginning of the next part of the 8080 code and sees a blank. Remember that this blank separates the opcode from its operand. We then skip to 4170 where we add spaces for pretty printing, and a tab for the Z80 code. Then we get the next subfield, which is the operand, and put it in N$. Back to 2150.

Here we jump immediately to 4380, which is a subroutine that puts all of our array data into another small array S1$(x). It leaves with X$ holding the Z80 code and the rest of the array filled with the subfields which do the translation of the operands. See lines 90 to 250 for a review of how these arrays and subfields are set up.

Now back to 2160 where we put our Z80 code into the buffers and add spaces and a tab.

The next job is to read the subfield codes so that the rest of the 8080 code can be properly added to the buffers. Our index L should contain the number of subfields there are. We start a loop that reads through the subfields and goes to the subroutine that produces a match. These subroutines are relatively simple.

The Subfield Subroutines

The <A> subroutine is the longest and most of the others jump into it, so we'll start our journey there at line 2800. The first thing we do is to get our next subfield from the 8080 code. Then we initialize H$ to the appropriate tree string and start to search the string for a match. The search routine at 3820 is simple enough, so I won't dwell on its finer points.

When we find a match we get the next subfield definition in the tree string, line 2920 to 2990, and edit it into the buffers, line 3000. We set our match flag if we hopefully did find it.

If you look at the flowcharts for the rest of the codes, you will see that <G>, <B>, <C>, <D>, and <E> all jump into the <A> subroutine, and that <F> and <S> do their own little thing. They are simple enough not to explain them in detail.

After the return from the subroutines, we end up back at 2370 and then 2410 on a good match. If there are more subfields to translate, we loop back to do the rest. We finally come to 2430 where we null out the small array, and then edit anything else from our 8080 code, such as comments, into our buffers. We end up at 2560 where we jump to 2660.

Get It Out

The routine here adds in our line number, prints it on the screen at 2670, adds a carriage return, and checks to see if our buffer plus the output buffer is less than our file buffer. If the buffer would overflow, it's time to put it onto disk, so we jump to the routine at 4710 that does just that. If not, we add the code to

the output buffer, and jump back to 1650 where we increment our line number and start the whole ball rolling again.

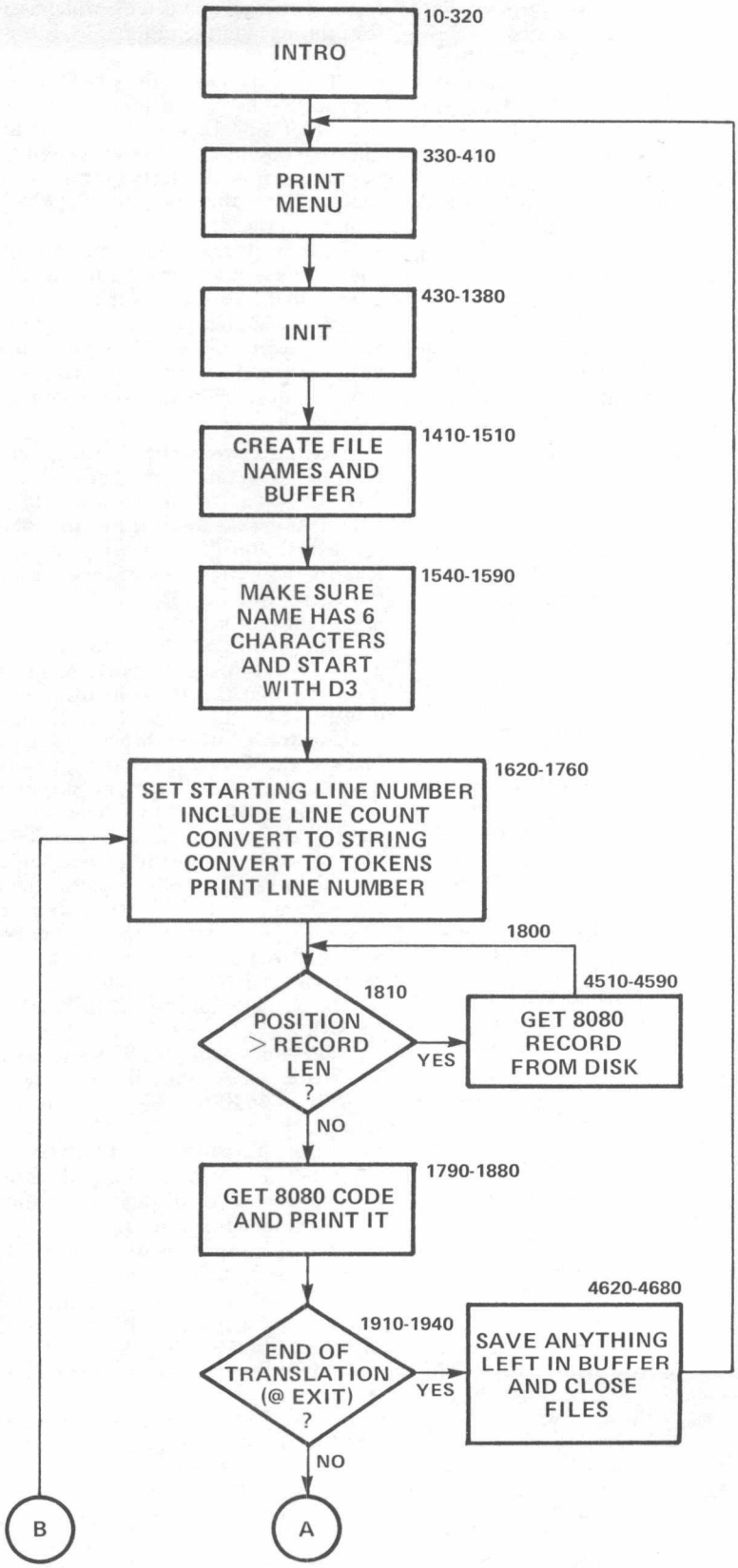
As you can see, all the routines are relatively simple; however, because of all the jumping around, it is difficult to understand and follow without a flowchart. This is a good reason to use more structured programming methods.

The program works, though, and if you need to get those old 8080 files into Z80 code, this is the program to use.

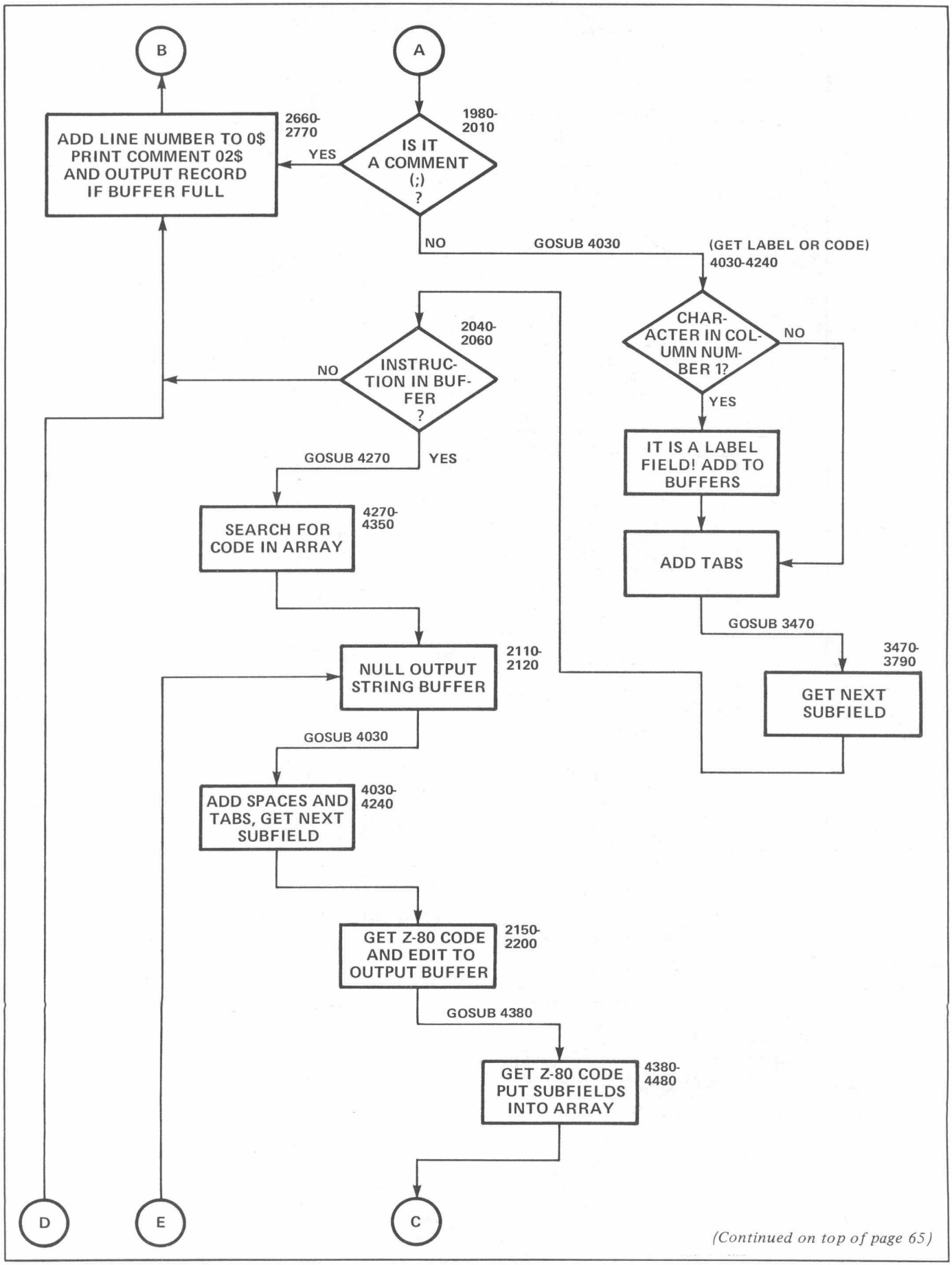
DDJ

(Flow charts from pages 63-67)

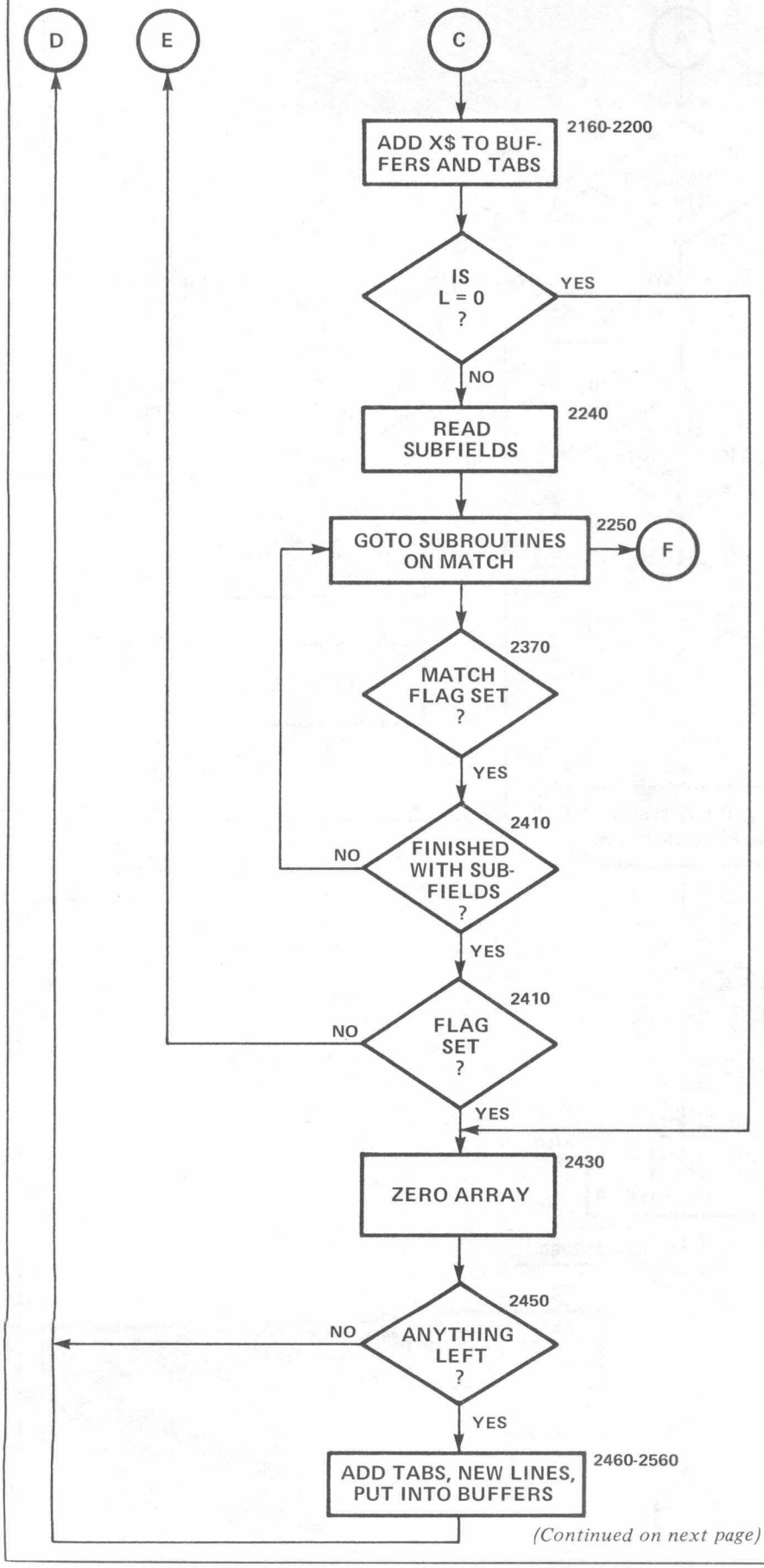
(Listing begins on page 68)



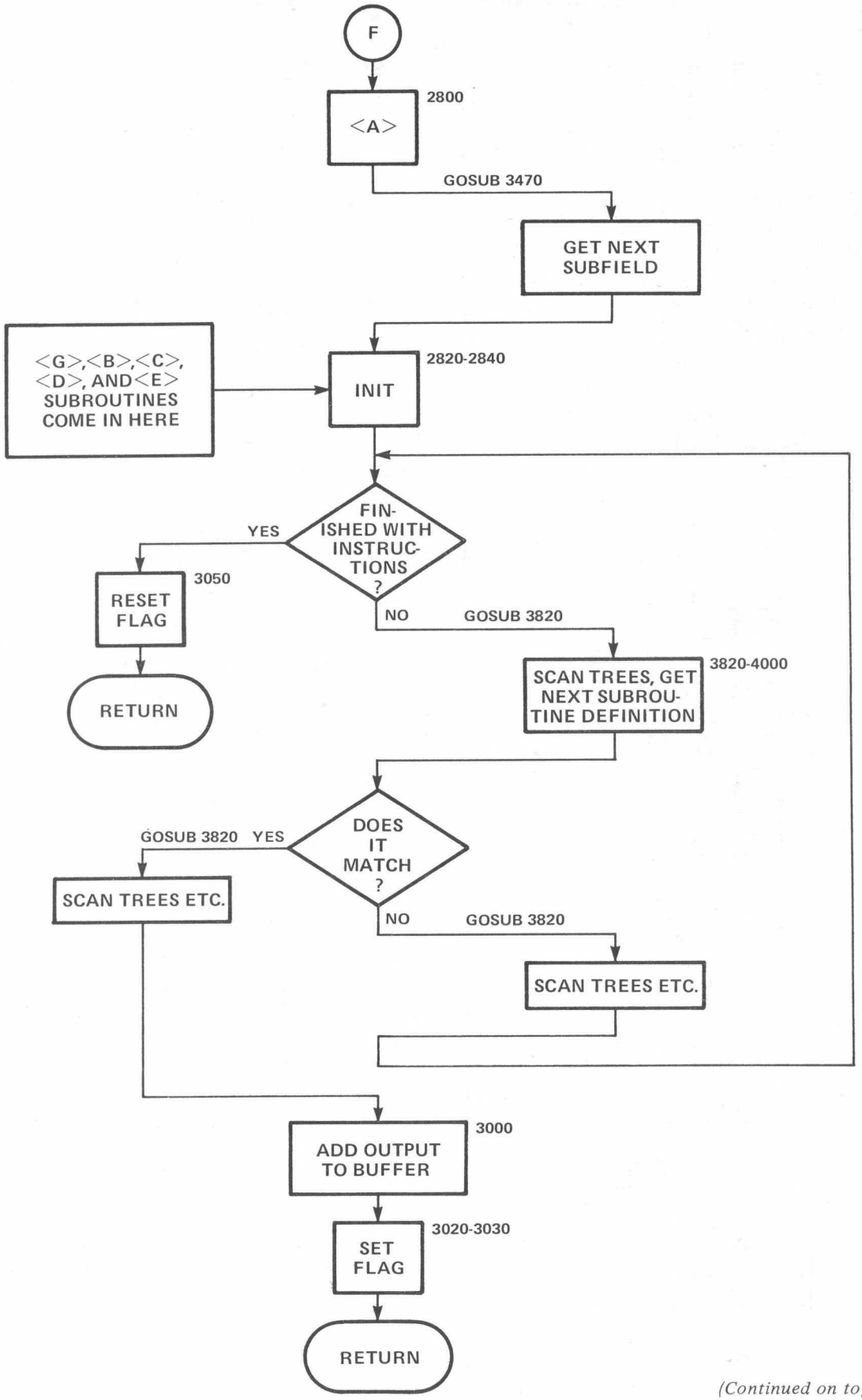
(Continued on next page)



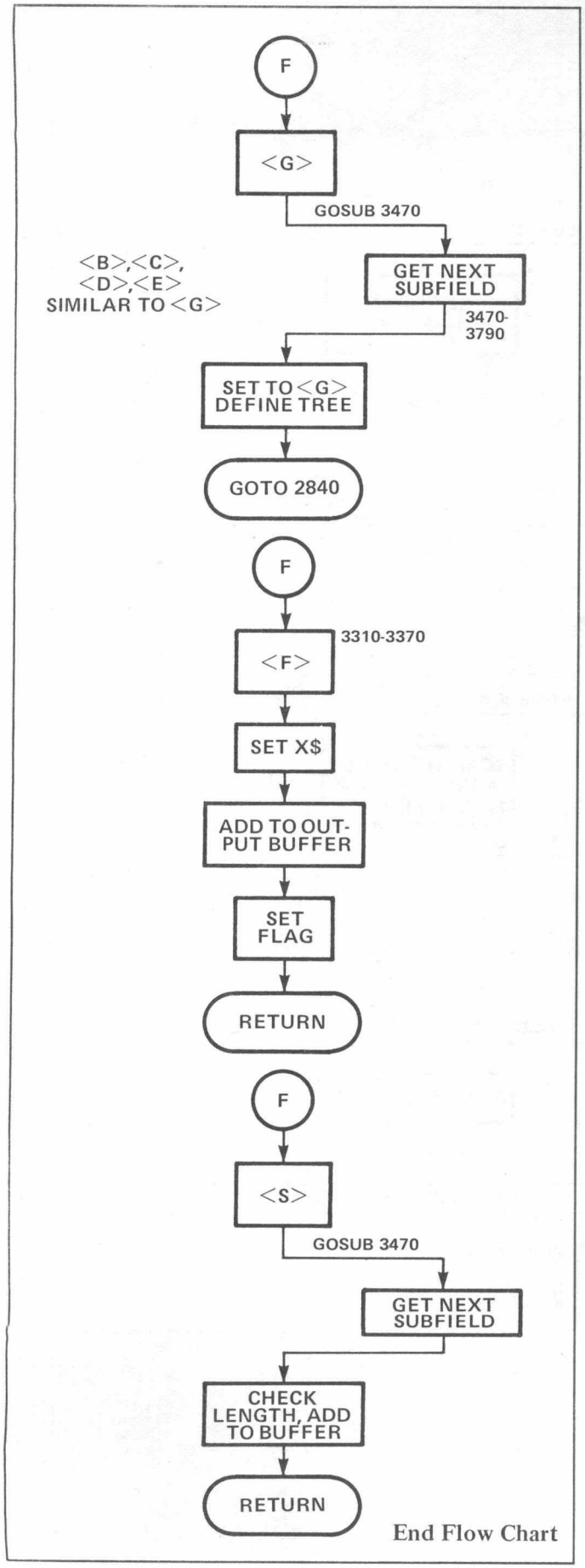
(Continued on top of page 65)



(Continued on next page)



(Continued on top of page 67)


F
<G>
GOSUB 3470
<B>,<C>,
<D>,<E>
SIMILAR TO <G>
GET NEXT SUBFIELD
3470-3790
SET TO <G> DEFINE TREE
GOTO 2840
F
3310-3370
<F>
SET X$
ADD TO OUT-PUT BUFFER
SET FLAG
RETURN
F
<S>
GOSUB 3470
GET NEXT SUBFIELD
CHECK LENGTH, ADD TO BUFFER
RETURN
End Flow Chart

8080 to Z80 Translator

(Text begins on page 60)

```
10 '8080 TO Z80 TRANSLATOR
20 '  FOR USE ON TRS-80, MODEL I, SINGLE DISK SYSTEM
30 '  USES ULTRADOS OPERATING SYSTEM
40 '  CREATES Z80 SOURCE FILE FOR EDITOR/ASSEMBLER
50 '  ORIGINAL VERSION BY ROBERT W. DEA, DDJ MAY80 #45 P.48
60 '  MODIFIED VERSION BY ANTHONY T. SCARPELLI, JUNE 1982
70 '
80 '
90 '  THE DATA IN THE ARRAY ARE DEFINED AS FOLLOWS:
100 '     L.8080MN.N.Z80MN.SUB1.SUB2....
110 ' L       = NUMBER OF CHARACTERS IN 8080 MNEMONIC
120 ' 8080MN = 8080 INSTRUCTION
130 ' N       = NUMBER OF SUBFIELDS
140 ' Z80MN   = Z80 EQUIVALENT INSTRUCTION
150 ' SUBN    = SUBFIELD DEFINITIONS
160 '
170 ' THE SUBFIELD DEFINITIONS ARE
180 '   <A> - REGULAR ONE TO ONE REGISTER TRANSLATION
190 '   <G> - RST NUMBER TRANSLATION
200 '   <B> - REGISTER PAIR TRANSLATION
210 '   <C> - REGISTER PAIR TO ACCUMULATOR
220 '   <D> - ACCUMULATOR TO REGISTER PAIR
230 '   <E> - EDIT A COMMA
240 '   <F> - EDIT FOLLOWING CHARACTER STRING TO OUTPUT
250 '   <S> - ONE TO ONE CHARACTER STRING TRANSLATION
260 '
270 '
280 '*** NOTE:  THIS VERSION USES A DOUBLE SPEED MODIFICATION
290 '           OUT 254,1 CAUSES DOUBLE SPEED CHANGEOVER
300 '           OUT 254,0 CAUSES NORMAL SPEED CHANGEOVER
310 '
320 '
330 CLS:PRINT"      **** 8080 TO Z80 TRANSLATOR ****"
340 PRINT:PRINT"ENTER 'C' TO CREATE AN 8080 SOURCE FILE"
350 PRINT"ENTER 'T' TO TRANSLATE 8080 TO Z80 CODE AND"
360 PRINT"           CREATE EDTASM SOURCE FILE"
370 PRINT"ENTER 'X' TO EXIT PROGRAM"
380 INPUT Q$
390 IF Q$="C" THEN 4820
400 IF Q$="X" THEN OUT 254,0:END
410 IF Q$<>"T" THEN 340
420 '
430 CLEAR 0:CLEAR 1500:DIM DA$(83)
440 ON ERROR GOTO 5320
450 ' MAKE SURE TWO BUFFERS ARE AVAILABLE (ULTRADOS)
460 IF PEEK(&521A)<2 THEN CMD"O":GOTO 460
470 '
480  DA$(1)="3.ACI.3.ADC.<F>.A,.<S>"
490  DA$(2)="3.ADC.3.ADC.<F>.A,.<A>"
500  DA$(3)="3.ADD.3.ADD.<F>.A,.<A>"
510  DA$(4)="3.ADI.3.ADD.<F>.A,.<S>"
520  DA$(5)="3.ANA.1.AND.<A>"
530  DA$(6)="3.ANI.1.AND.<S>"
540  DA$(7)="4.CALL.1.CALL.<S>"
550  DA$(8)="2.CC.3.CALL.<F>.C,.<S>"
560  DA$(9)="2.CM.3.CALL.<F>.M,.<S>"
570  DA$(10)="3.CMA.0.CPL"
580  DA$(11)="3.CMC.0.CCF"
590  DA$(12)="3.CMP.1.CP.<A>"
600  DA$(13)="3.CNC.3.CALL.<F>.NC,.<S>"
610  DA$(14)="3.CNZ.3.CALL.<F>.NZ,.<S>"
620  DA$(15)="2.CP.3.CALL.<F>.P,.<S>"
630  DA$(16)="3.CPE.3.CALL.<F>.PE,.<S>"
640  DA$(17)="3.CPI.1.CP.<S>"
650  DA$(18)="3.CPO.3.CALL.<F>.PO,.<S>"
660  DA$(19)="2.CZ.3.CALL.<F>.Z,.<S>"
670  DA$(20)="3.DAA.0.DAA"
680  DA$(21)="3.DAD.3.ADD.<F>.HL,.<B>"
690  DA$(22)="2.DB.1.DEFB.<S>"
700  DA$(23)="3.DCR.1.DEC.<A>"
710  DA$(24)="3.DCX.1.DEC.<B>"
720  DA$(25)="2.DI.0.DI"
730  DA$(26)="2.DM.1.DEFM.<S>"
```

(Continued on page 70)

8080 to Z80 Translator

(Listing continued, text begins on page 60)

```
740  DA$(27)="2.DW.1.DEFW.<S>"
750  DA$(28)="2.EI.0.EI"
760  DA$(29)="3.EQU.1.EQU.<S>"
770  DA$(30)="3.HLT.0.HALT"
780  DA$(31)="2.IN.5.IN.<F>.A,(.<S>.<F>.)."
790  DA$(32)="3.INR.1.INC.<A>"
800  DA$(33)="3.INX.1.INC.<B>"
810  DA$(34)="2.JC.3.JP.<F>.C,.<S>"
820  DA$(35)="2.JM.3.JP.<F>.M,.<S>"
830  DA$(36)="3.JMP.1.JP.<S>"
840  DA$(37)="3.JNC.3.JP.<F>.NC,.<S>"
850  DA$(38)="3.JNZ.3.JP.<F>.NZ,.<S>"
860  DA$(39)="2.JP.3.JP.<F>.P,.<S>"
870  DA$(40)="3.JPE.3.JP.<F>.PE,.<S>"
880  DA$(41)="3.JPO.3.JP.<F>.PO,.<S>"
890  DA$(42)="2.JZ.3.JP.<F>.Z,.<S>"
900  DA$(43)="3.LDA.5.LD.<F>.A,(.<S>.<F>.)"
910  DA$(44)="4.LDAX.1.LD.<D>"
920  DA$(45)="4.LHLD.5.LD.<F>.HL,(.<S>.<F>.)"
930  DA$(46)="3.LXI.3.LD.<B>.<E>.<S>"
940  DA$(47)="3.MOV.3.LD.<A>.<E>.<A>"
950  DA$(48)="3.MVI.3.LD.<A>.<E>.<S>"
960  DA$(49)="3.NOP.0.NOP"
970  DA$(50)="3.ORA.1.OR.<A>"
980  DA$(51)="3.ORG.1.ORG.<S>"
990  DA$(52)="3.ORI.1.OR.<S>"
1000  DA$(53)="3.OUT.5.OUT.<F>.(.<S>.<F>.),A"
1010  DA$(54)="4.PCHL.2.JP.<F>.(HL)"
1020  DA$(55)="3.POP.1.POP.<B>"
1030  DA$(56)="4.PUSH.1.PUSH.<B>"
1040  DA$(57)="3.RAL.0.RLA"
1050  DA$(58)="3.RAR.0.RRA"
1060  DA$(59)="2.RC.2.RET.<F>.C"
1070  DA$(60)="3.RET.0.RET"
1080  DA$(61)="3.RLC.0.RLCA"
1090  DA$(62)="2.RM.2.RET.<F>.M"
1100  DA$(63)="3.RNC.2.RET.<F>.NC"
1110  DA$(64)="3.RNZ.2.RET.<F>.NZ"
1120  DA$(65)="2.RP.2.RET.<F>.P"
1130  DA$(66)="3.RPE.2.RET.<F>.PE"
1140  DA$(67)="3.RPO.2.RET.<F>.PO"
1150  DA$(68)="3.RRC.0.RRCA"
1160  DA$(69)="3.RST.1.RST.<S>"
1170  DA$(70)="2.RZ.2.RET.<F>.Z"
1180  DA$(71)="3.SBB.3.SBC.<F>.A,.<A>"
1190  DA$(72)="3.SBI.3.SBC.<F>.A,.<S>"
1200  DA$(73)="4.SHLD.5.LD.<F>.(.<S>.<F>.),HL"
1210  DA$(74)="4.SPHL.2.LD.<F>.SP,HL"
1220  DA$(75)="3.STA.5.LD.<F>.(.<S>.<F>.),A"
1230  DA$(76)="4.STAX.1.LD.<C>"
1240  DA$(77)="3.STC.0.SCF"
1250  DA$(78)="3.SUB.1.SUB.<A>"
1260  DA$(79)="3.SUI.1.SUB.<S>"
1270  DA$(80)="4.XCHG.2.EX.<F>.DE,HL"
1280  DA$(81)="3.XRA.1.XOR.<A>"
1290  DA$(82)="3.XRI.1.XOR.<S>"
1300  DA$(83)="4.XTHL.2.EX.<F>.(SP),HL"
1310 '
1320 ' INITIALIZE SYNTAX TREE STRINGS
1330 A3$="A A B B C C D D E E H H L L M (HL)"
1340 B$="PSW AF B BC D DE H HL SP SP"
1350 C$="B (BC),A D (DE),A"
1360 D$="B A,(BC) D A,(DE)"
1370 E$=" , ,"
1380 G$="0 0 1 8 2 10H 3 18H 4 20H 5 28H 6 30H
      7 38H"
1390 '
1400 ' CREATE FILE NAMES AND BUFFERS
1410 PRINT:INPUT"ENTER 8080 FILE NAME ";I9$
1420 OUT 254,0:  'NORMAL SPEED
1430 OPEN "R",1,I9$
1440 FIELD 1, 255 AS A$
1450 INPUT"ENTER Z80 DISK FILESPEC ";F9$
```

```
1460 INPUT"ENTER EDTASM SOURCE FILE NAME (6 CHAR. MAX) ";O9$
1470 IF LEN(O9$)>6 THEN PRINT"** TOO LONG **":GOTO 1460
1480 OPEN "R",2,F9$
1490 FIELD 2, 255 AS Z$
1500 OUT 254,1   ' *** DOUBLE SPEED
1510 R=1:F1=1:W=1:C9=0
1520 '
1530 ' FILL IN FILE NAME WITH SPACES IF NECESSARY
1540 K=LEN(O9$)
1550 IF K=6 THEN 1590
1560 FOR I=K TO 5
1570   O9$=O9$+" "
1580 NEXT I
1590 O1$=CHR$(211)+O9$
1600 '
1610 ' GENERATE AUTO LINE NUMBERING & INC BY 10
1620 INPUT"START LINE NUMBERS AT ";C0:PRINT
1630 C0=C0-10:C1=900000+C0
1640 ' INC LINE COUNT BY 10
1650 C1=C1+10:C3$=""
1660 ' CONVERT TO ASCII NUMBER 5 PLACES
1670 C1$=STR$(C1)
1680 C1$=RIGHT$(C1$,5)+" "
1690 ' CONVERT TO TOKENS
1700 FOR I=1 TO 5
1710   C4$=CHR$(VAL(MID$(C1$,I,1))+176)
1720   C3$=C3$+C4$
1730 NEXT I
1740 C2$=C1$
1750 C3$=C3$+" "
1760 PRINT"- ";C2$+" - ";
1770 '
1780 ' GET NEXT 8080 INSTRUCTION
1790 Z1$=CHR$(91)
1800 R1$=""
1810 IF F1>C9 THEN 4510
1820 R1$=R1$+MID$(R$,F1,1)
1830 IF RIGHT$(R1$,1)=Z1$ THEN 1860
1840 F1=F1+1
1850 GOTO 1810
1860 F1=F1+1
1870 I$=LEFT$(R1$,LEN(R1$)-1)
1880 PRINT I$
1890 '
1900 ' DO WE HAVE END OF TRANSLATION?
1910 IF I$<>"@EXIT" THEN 1980
1920 PRINT:PRINT"IF NOT END OF LISTING NOTE LINE NUMBER."
1930 INPUT"PRESS <ENTER> TO CLOSE FILE AND RETURN TO MENU";X$
1940 GOTO 4620
1950 '
1960 ' EDIT LINE# IN AND CONVERT TO Z80 INSTRUCTION
1970 ' CHECK FOR JUST COMMENT LINE
1980 IF LEFT$(I$,1)<>";"THEN 2010
1990 O$=O$+I$:O2$=O2$+I$
2000 GOTO 2660
2010 GOSUB 4030
2020 '
2030 ' DO WE HAVE AN 8080 INSTRUCTION IN INSTRUCTION BUFFER?
2040  IF LEN(N$)=0 THEN 2660
2050 ' YES! NOW SEARCH FOR MATCH
2060  GOSUB 4270
2070 '
2080 ' THERE WAS A MATCH, Z8$=8080 CODE
2090 '  GET NEXT STRING
2100 ' INIT OUTPUT STRING BUFFER
2110  O$="":O2$=""
2120  GOSUB 4030
2130 '
2140 ' NOW GET Z80 OUTPUT INSTR AND EDIT TO OUTPUT BUFFERS
2150 GOSUB 4380
2160 O$=O$+X$:O2$=O2$+X$
2170 FOR K=LEN(O2$)+1 TO 16
2180   O2$=O2$+" "
2190 NEXT K
2200 O$=O$+CHR$(09)
2210 '
2220 ' NOW READ TRANSLATION SUBFIELDS
2230 IF L=0 THEN 2430
```

(Continued on next page)

8080 to Z80 Translator

(Listing continued, text begins on page 60)

```
2240 FOR I=2 TO L+1
2250    S$=S1$(I)
2260   ' CHECK SUBSET FOR DEFINITION MATCH AND TRANSFER TO
2270   '   DEFINITION PROCESS SECTION.
2280    IF S$="<A>" THEN GOSUB 2800
2290    IF S$="<G>" THEN GOSUB 3100
2300    IF S$="<B>" THEN GOSUB 3150
2310    IF S$="<C>" THEN GOSUB 3190
2320    IF S$="<D>" THEN GOSUB 3230
2330    IF S$="<E>" THEN GOSUB 3270
2340    IF S$="<F>" THEN GOSUB 3310
2350    IF S$="<S>" THEN GOSUB 3390
2360   ' IS MATCH FLAG SET?
2370    IF F=1 THEN 2410
2380   ' WE GET HERE IF PARSE FAILED
2390   GOTO 2110
2400   ' YES WE HAVE A GOOD TRANSLATION
2410   NEXT I:IF F<>1 THEN 2110
2420 ' ZERO ARRAY
2430 FOR I=1 TO 6:S1$(I)="":NEXT I
2440 ' EDIT ANYTHING THATS LEFT
2450  IF T+1>LEN(I$) THEN 2560
2460  FOR K=LEN(O2$)+1 TO 29
2470    O2$=O2$+" "
2480  NEXT K
2490  O$=O$+CHR$(09)+CHR$(09)
2500  FOR K=1 TO LEN(I$)
2510    T1$=MID$(I$,K,1)
2520    IF T1$=";" THEN GOTO 2540
2530  NEXT K
2540 O$=O$+RIGHT$(I$,LEN(I$)-K+1)
2550 O2$=O2$+RIGHT$(I$,LEN(I$)-K+1)
2560  GOTO 2660
2570 '
2580 ' NO FIND EXIT
2590 O2$="*NO MATCH* "+I$
2600 ' OUTPUT *NO MATCH* MSG AND DEC LINE COUNTER BY 10
2610 PRINT O2$
2620 C1=C1-10
2630 O$="":GOTO 2690
2640 '
2650 ' EDIT LINE# IN AND OUTPUT THE TRANSLATED Z80 INST
2660  O$=C3$+O$
2670  PRINT C1$+O2$
2680  O$=O$+CHR$(13)
2690  L9=LEN(O1$)+LEN(O$)
2700  IF L9<255 THEN 2750
2710  L8=255-LEN(O1$)
2720  O1$=O1$+LEFT$(O$,L8)
2730  O$=RIGHT$(O$,LEN(O$)-L8)
2740  GOTO 4710
2750  O1$=O1$+O$
2760  O$="":O2$=""
2770  GOTO 1650
2780 '
2790 ' PROCESS SINGLE REG SECTION "<A>"
2800 GOSUB 3470
2810 ' INIT PROCESS SECTION AND SET TO <A> TREE
2820 H$=A3$
2830 ' INIT TREE SEARCH SECTION
2840 H2=0:T2=1
2850 ' CHECK IF WE PARSED THE WHOLE INPUT 8080 INST YET.
2860 ' SEE IF IT SATISFIES THE 8080 DEFINITION
2870 IF T2>LEN(H$) THEN 3050
2880 ' GO SCAN THE DEFINITION TREE FROM LEFT TO RIGHT TO
2890 ' GET NEXT SUB DEFINITION
2900 GOSUB 3820
2910 ' DOES IT MATCH THE SUB ELEMENT WE HAVE?
2920 IF M$=N$ THEN 2990
2930 ' NO IT DOES NOT, SO SKIP THE NEXT Z80 SUB TRANSLATION
2940 '   AND GO TRY FOR ANOTHER 8080 SUB DEFINITION MATCH.
2950 GOSUB 3820
2960 GOTO 2870
```

```
2970 ' WE HAVE AN 8080 SUB DEFINITION MATCH. NOW GET THE
2980 '    Z80 TRANSLATION AND EDIT IT INTO THE OUTPUT BUFFER O$.
2990 GOSUB 3820
3000 O$=O$+M$:O2$=O2$+M$
3010 ' SET MATCH FLAG
3020 F=1
3030 RETURN
3040 ' SET NO MATCH FLAG
3050 F=0
3060 RETURN
3070 '
3080 ' PROCESS "<G>" DEFINITIONS
3090 ' GET NEXT INPUT SUB FIELD
3100 GOSUB 3470
3110 ' SET TO <G> DEFINITION TREE
3120 H$=G$
3130 GOTO 2840
3140 ' PROCESS "<B>" DEF
3150 GOSUB 3470
3160 H$=B$
3170 GOTO 2840
3180 ' PROCESS "<C>" DEF
3190 GOSUB 3470
3200 H$=C$
3210 GOTO 2840
3220 ' PROCESS "<D>" DEF
3230 GOSUB 3470
3240 H$=D$
3250 GOTO 2840
3260 ' PROCESS "<E>" DEF
3270 GOSUB 3470
3280 H$=E$

3290 GOTO 2840
3300 ' PROCESS "<F>" DEF
3310 X$=S1$(I+1)
3320 ' JUST EDIT NEXT Z80 TRANS ELEMENT TO OUTPUT BUFFER
3330 O$=O$+X$:O2$=O2$+X$
3340 ' SET MATCH FLAG
3350 F=1
3360 I=I+1
3370 RETURN
3380 ' PROCESS "<S>" DEF
3390 GOSUB 3470
3400 F=1
3410 IF LEN(N$)=0 THEN F=0
3420 O$=O$+N$:O2$=O2$+N$
3430 RETURN
3440 '
3450 ' SCANNER RETURNS N$
3460 ' INIT SUB FIELD SCANNER SECTION
3470 H=T+1
3480 N$=""
3490 H1=H
3500 ' SCAN FOR NEXT 8080 SUB FIELD STARTING WITH FIRST
3510 '    NON-BLANK.
3520 FOR J=H1 TO LEN(I$)
3530    ' GET NEXT CHAR
3540    ' IS IT A BLANK?
3550    IF MID$(I$,J,1)<>" " THEN J=LEN(I$):GOTO 3590
3560    ' YES! GO GET NEXT CHAR
3570    H=H+1
3580    T=T+1
3590 NEXT J
3600 ' WE SHOULD BE POINTING TO START OF 8080 SUB FIELD
3610 H1=H
3620 ' SCAN UNTIL WE GET A TERMINATOR OR END OF LINE.
3630 ' ("BLANK" AND "," ARE TERMINATORS)
3640 FOR J=H1 TO LEN(I$)
3650    ' GET NEXT CHAR
3660    ' IS IT A BLANK?
3670    IF MID$(I$,J,1)=" " THEN GOTO 3790
3680    ' IS IT A ","
3690    IF MID$(I$,J,1)="," THEN GOTO 3770
3700    ' NO! ITS NONE OF THE ABOVE SO EDIT THIS CHAR INTO
3710    '    INSTRUCTION BUFFER.
3720    N$=N$+MID$(I$,J,1)
3730    T=T+1
3740    ' GO GET NEXT CHAR
```

(Continued on next page)

8080 to Z80 Translator

(Listing continued, text begins on page 60)

```
3750 NEXT J
3760 RETURN
3770 IF LEN(N$)=0 THEN T=T+1
3780 IF LEN(N$)=0 THEN N$=","
3790 RETURN
3800 '
3810 ' SCAN THE TREES
3820 H2=T2
3830 M$=""
3840 H3=H2
3850 FOR J=H3 TO LEN(H$)
3860   ' TAB TO NEXT NON-BLANK CHAR
3870   IF MID$(H$,J,1)<>" "THEN J=LEN(H$):GOTO 3900
3880   H2=H2+1
3890   T2=T2+1
3900 NEXT J
3910 H3=H2
3920 ' GET NEXT SUB DEF OF 8080 INSTRUC.
3930 FOR J=H3 TO LEN(H$)
3940   ' HAVE WE A BLANK YET?
3950   IF MID$(H$,J,1)=" " THEN J=LEN(H$):GOTO 3990
3960   ' NO! THEN ADD THIS CHAR
3970   M$=M$+MID$(H$,J,1)
3980   T2=T2+1
3990 NEXT J
4000 RETURN
4010 '
4020 ' SECTION TO EDIT A LABEL AND OUTPUT Z80 TRANSLATION
4030 H=0:T=1
4040 ' IS THERE A CHAR IN COL#1?
4050 IF LEFT$(I$,1)=" " THEN 4170
4060 H=1
4070 FOR K=1 TO LEN(I$)
4080   ' GET NEXT CHAR
4090   ' IS IT A BLANK
4100   IF MID$(I$,K,1)=" " THEN K=LEN(I$):GOTO 4150
4110   ' NO! NOT A BLANK THEN MUST BE PART OF LABEL FIELD.
4120   ' SO EDIT IT INTO OUTPUT BUFFER
4130   O$=O$+MID$(I$,K,1):O2$=O2$+MID$(I$,K,1)
4140   T=K+1
4150 NEXT K
4160 ' TAB TO COL 8 FOR PRINTOUT
4170 FOR K=LEN(O2$)+1 TO 8
4180   O2$=O2$+" "
4190 NEXT K
4200 ' INSERT TAB INTO OUTPUT BUFFER
4210 O$=O$+CHR$(09)
4220 ' NOW SCAN FOR THE 8080 INSTRUC.
4230 GOSUB 3470
4240 RETURN
4250 '
4260 ' BINARY SEARCH THROUGH DATA STATEMENTS FOR MATCH
4270 U=83:L=1
4280 F=INT((L+U)/2)   ' CALCULATE FENCE
4290 N=VAL(LEFT$(DA$(F),1)):Z8$=MID$(DA$(F),3,N)
4300 IF Z8$=N$ THEN RETURN
4310 IF L>=U THEN 2590
4320 IF Z8$>N$ THEN 4340
4330 L=F+1:GOTO 4280
4340 U=F-1:GOTO 4280
4350 GOTO 2590   ' NO MATCH
4360 '
4370 ' SUBROUTINE TO READ THROUGH ARRAY DATA
4380 L=VAL(LEFT$(DA$(F),1)):N1=L+4
4390 L=VAL(MID$(DA$(F),N1,1))
4400 FOR I=1 TO L+1
4410   FOR J=1 TO 6
4420     S1$=MID$(DA$(F),N1+1+J,1)
4430     IF S1$="." THEN N1=N1+J:J=6:GOTO 4450
4440     S1$(I)=S1$(I)+S1$
4450   NEXT J
4460 NEXT I
4470 X$=S1$(1)
```

```
4480 RETURN
4490 '
4500 ' GET RECORD FROM DISC
4510 R$=""
4520 OUT 254,0
4530 GET 1,R
4540 R$=A$
4550 F1=1
4560 R=R+1
4570 C9=LEN(R$)
4580 OUT 254,1    '*** DOUBLE SPEED
4590 GOTO 1800
4600 '
4610 ' CLOSE FILES
4620 IF LEN(O1$)<0 THEN OUT 254,0:GOTO 4670
4630 O1$=O1$+CHR$(26)
4640 LSET Z$=O1$
4650 OUT 254,0
4660 PUT 2,W
4670 CLOSE 1:CLOSE 2
4680 GOTO 330
4690 '
4700 ' PUT RECORD ONTO DISC
4710 LSET Z$=O1$
4720 OUT 254,0
4730 PUT 2,W
4740 W=W+1
4750 O1$=""
4760 'OUT 254,1
4770 GOTO 2750
4780 '
4790 '
4800 '
4810 ' SECTION TO CREATE AN 8080 SOURCE FILE
4820 '
4830 CLEAR 0:CLEAR 1500:CMD"O"   ' MAKE SURE BUFFER AVAILABLE
4840 R=1:A$=""
4850 '
4860 CLS:PRINT"      *  8080 CREATE PROGRAM  *"
4870 PRINT
4880 PRINT"TYPE 8080 SOURCE IN FOLLOWING FORMAT:"
4890 PRINT"  LABEL - OPERATION - OPERAND - COMMENT"
4900 PRINT"  EXAMPLE:  START LXI H,0001H ;LOAD ADDRESS
4910 PRINT"               XRA A ;CLEAR A REGISTER"
4920 PRINT"  ONLY ONE SPACE IS REQUIRED BETWEEN ENTRIES."
4930 PRINT"A LABEL OR LEADING ';' SHOULD NOT START WITH"
4940 PRINT"  A SPACE, AN OPERATION SHOULD."
4950 PRINT"AN 'N' AFTER THE 'SURE' WILL CANCEL ENTRY,"
4960 PRINT"  AN <ENTER> ADDS ENTRY TO FILE."
4970 PRINT"TYPE '@EXIT' TO EXIT TO MENU."
4980 PRINT"A BUFFER FROM 215 TO 255 CHARACTERS LONG IS"
4990 PRINT"  SAVED TO DISK."
5000 PRINT:INPUT"ENTER 8080 OUTPUT FILE NAME ";R$
5010 PRINT"OUTPUT FILE '";R$;"' IS ON DRIVE #0"
5020 A3$="@EXIT"+CHR$(91)
5030 OPEN "R",1,R$
5040 FIELD 1,255 AS A$
5050 PRINT
5060 X$=""
5070 PRINT"    ";CHR$(92)
5080 LINE INPUT"--> "; A1$
5090 B=LEN(A1$)+1:C=LEN(A2$)
5100 IF B+C>255 THEN PRINT"NEXT ENTRY WILL EXCEED
 BUFFER SPACE AND PRODUCE ERROR!!   SAY 'N'!
 YOU HAVE ONLY";255-LEN(A2$);" CHARACTERS LEFT!"
5110 INPUT"SURE";X$
5120 IF X$="N" THEN 5060
5130 A1$=A1$+CHR$(91)
5140 A=LEN(A2$)
5150 IF A>215 THEN 5190
5160 A2$=A2$+A1$
5170 IF A1$=A3$ THEN 5200
5180 GOTO 5060
5190 IF LEN(A1$)<1 THEN 5240
5200 LSET A$=A2$
5210 PUT 1,R
5220 R=R+1
5230 IF A1$=A3$ THEN 5270 ELSE 5250
5240 A1$=""
```

(Continued at right, top of page)

```
5250 A2$=A1$
5260 GOTO 5060
5270 CLOSE 1
5280 GOTO 330
5290 END
5300 '
5310 ' ERROR TRAP
5320 OUT 254,0
5330 RESUME
5340 '
5350 ' END OF PROGRAM
```

End Listing

DR. DOBB'S CLINIC

by D. E. Cortesi

Disk Drives Us Crazy

In February we presented Loren Amelang's request for explanations why perfectly good diskettes could come up with read errors after being stored for a while. David Oster (whose letterhead proclaims The People's Republic of Santa Monica, its motto "Unity and Breakfast for All") suggests two ways. First, he points out, drives vary with time. "Apple drives, in particular, seem to need frequent realignment. The disks you read and write every day change with the drive, but the drive changes enough that stored disks stop working."

Amelang also wondered why one program could always read the bad diskettes when others couldn't. Oster has a comment on this, too.

"When I had this problem it turned out that the programs were accessing the disk in very different ways. When the program that failed wanted to go to an inner track like track 70, it would tell the controller "go to track 70." When the program that worked wanted to go there, it told the controller "go to the next track," 70 times. If you tell a disk controller to step once, it just does it. If you tell it to seek a long way, it runs an internal subroutine that does a fancy acceleration-deceleration of the head carriage. Only when it gets there does it begin looking for a track. If the disk is a slightly different shape than the controller expects, the position error in going 70 tracks is 70 times as large as the position error from stepping one track.

"I had this problem because my drives weren't properly cooled. There was a nice big fan in the cabinet, but there were a couple of unused connector holes and all the air flowed through them and not enough of it flowed over the electronics. I replaced a few heat-damaged chips, covered the extra holes with tape, and haven't had a problem since."

OK, those are reasonable propositions (especially the part about air flow), but they sound to us as if they would be *hardware-specific*. For example, according to its data sheets, the Western Digital FD1793 controller (used in the popular California Computer Systems 2422 board) performs a seek from track zero to track 70 by doing 70 "step-in" operations. And the 1793 doesn't make any adjustments on the resulting position; it just assumes that a step is a step.

There must be a lot more things that could be said about Loren Amelang's problem. Another open problem is that of Ernest Knipp, whose Z80 system occasionally goes catatonic. We look forward to more contributions on both these topics.

Counts and Measures

In that February issue, we gave Burks Smith room to speak out on the use of floating-point arithmetic, and its lack of precision as usually implemented. The heart of his presentation may have been his statement that "If something called 'double-precision arithmetic' can't take ten percent of a dollar and get *exactly* ten cents, there shouldn't be any excuses. The answer is just plain wrong!" Oster has a comment on that.

"Smith has confused *counts* with *measures*. When you say you have ten dollars, you are saying you have 1,000 pennies – a count. When you say you have ten gallons of water, you are not saying that you have a certain number of water molecules, but making a statement about a physical quantity that you only know to a certain precision – a measure. Floating-point numbers were designed for calculation with measures; integers were designed for calculation with counts.

"So, if something called 'double-precision arithmetic' takes ten percent of a dollar, it is right as long as its answer lies between 0.999...9 and 1.000...01. It may not be what you *want*, but it is what that arithmetic is designed to do. There are some experimental computer languages where all numbers input to a program must be labeled with whether they are measures or counts, and if measures, how accurate they are and what units they are in. The results of the program are automatically expressed in appropriate units and you are told how precise the results are."

Distinguishing "counts" and "measures" certainly helps clarify the problem in our mind, at least. On reflection, it seems that the distinction we try to draw between "commercial" and "scientific" applications is based on which kind of numbers is preponderant in each. What we call "commercial" work deals mostly with counts; "scientific" computing deals mostly with measures. Think about it.

But there is more to be said on this topic as well, and your contribution is still welcome. We asked other questions in that February column. CBASIC uses a BCD representation for float numbers that has about the same precision as MBASIC's binary one – has anyone really compared the results of the two on the same problem set? We also wondered if there were any ratios that have a finite representation in binary but not in decimal (as there are decimal fractions that can't be represented in binary). Oster's answer is "no"; can you figure out what leads him to this conclusion?

Spotting DDT

Back in October of 1982, we showed one way to get DDT to load in such a way that the resident parts of CP/M were still visible and the CCP wasn't overlaid. Aubrey Hutchison has another way. If you know where you want DDT to load itself, the most direct way is to alter its code. Use DDT to modify DDT.COM. Look at the instructions around 0150h; you should find

```
0150 SUB B
0151 MOV D,A
```

If you replace those two instructions with the single instruction

```
0150 MVI D,xx
```

you will have a version of DDT that will always load itself at *xx*00h in storage.

Fun With Your New BIOS

We're working at bringing up CP/M 3 (or "CP/M Plus," as Digital Research wants to call it) on our own system. The first step is to get the present BIOS to assemble with a relocating assembler. We're using RMAC and LINK80 from Digital Research, mostly because RMAC allows the long, long labels that we like to use.

Rather than do a whole lot of new code, then debug it while learning a new operating system, we thought we would first make the new BIOS work with CP/M 2.2. Then we'd enhance it, still on 2.2, and finally make the few changes needed for CP/M 3. One of the enhancements we want to make while still running the old operating system is to split the BIOS into a fixed part and a banked part. The more complicated disk functions go into the banked BIOS; the serial I/O and the trivial disk functions like "set track" remain in the fixed part.

With the complicated functions set aside in banked storage, we can really let them spread out and get competent. For example, we plan to display disk error messages in detail, in English. That's something there just isn't room for when the whole BIOS has to be squeezed into a

couple of kilobytes. Another planned enhancement is the use of separate read and write sector buffers for each physical drive, so when programs alternate reading and writing they won't force the same big sectors to be read over and over. Still another is the mapping of four logical drives onto two physical ones, as is done in the IBM PC. Doing this and maintaining the full generality of use for all four drives turns out to be trickier than we thought, but it's possible when a few hundred additional bytes don't matter.

Fun With LINK80

OK, so you plan a much larger BIOS. Obviously you don't want to keep it as one big assembly; it's too difficult to edit and too tedious to assemble that way. You want the BIOS broken into lots of small modules, each with a well-defined set of functions (we ended up with a dozen, five fixed and seven banked). It's one of the great advantages of having a linker that you can do that easily, right?

Wrong. It isn't easy at all with LINK-80, because a two-part BIOS is not the sort of program it was designed to link. It doesn't help that the manual addresses the use of the program only as an adjunct to PL/I-80 and doesn't explain it as a general tool. We managed it in the end and these are some of the things we learned along the way.

First, the "L" option determines the origin of the linked program *and* the starting point of the linker's output file. If you were linking a (non-banked) BIOS that ran at F200h, you would do it like this:

```
LINK80 BIOS=modules. . .[LF200]
```

That produces BIOS.COM, containing just the code of the given modules, linked to an origin of F200h. The same effect can be had with the "P" option, but the resulting file contains enough binary zeroes to fill storage from 0100h to the program origin – about 61K of them in the example. We can't figure out what the "P" option is good for.

Next, the "D" option determines the origin of the data segments of the linked modules, while the "L" option sets the origin of the code segments. We coded our banked BIOS so that the banked modules consist entirely of data segments and the fixed modules are entirely code segments. We casually assumed that a command like

```
LINK80 BIOS=modules. . .
    [D4000,LF200]
```

would link the banked parts to an origin of 4000h and the fixed parts at F200h. It doesn't. It produces a file that contains only the code segments, linked to F200h. The data segments vanish. In fact, LINK80 will never write any output that would fall below the "L" load-point, whatever segment it comes from. The linking is done correctly in that external references have correct values, but the data they point to is omitted from the file if it falls below the "L" address.

That's just as well, because we weren't thinking too clearly when we devised the command above. What we really want is two separate files, FIX.COM and BNK.COM, the first containing the code that is to go into high, global storage and the second containing code to go into the banked storage. That implies that LINK80 has to process each group of modules separately. But each section has references to public labels in the other. That implies that LINK80 has to process all modules together in one run so that it can resolve the interlocking references. If this were a movie, our computer would be saying, "Contradiction, does not compute" in a hollow voice. CP/M 3 is supposed to have a command that does all this, but we are working under CP/M 2.2.

Enter the Overlay

Just as we were about to give up, we found the appendix on overlays in the LINK80 manual. They aren't explained very well, presumably because users of PL/I-80 aren't supposed to be concerned about the details. But in fact, LINK80's overlay feature can give us just what we need for the present problem.

To LINK80, an overlay is another program, one that just happens to be linked at the same time as some main program. The syntax that specifies an overlay is identical to the syntax for a main program, but given recursively in parentheses. The command lines could get out of hand, except that they can be continued by appending an ampersand. Here is how a PL/I user might link a program with overlays:

```
LINK80 ROOT=modules. . .        &
    (OV1=modules. . .)          &
    (OV2=modules. . .           &
        (OV2A=modules. . .)     &
        (OV2B=modules. . .)      )
```

That command would create five output files: ROOT.COM, OV1.OVL, OV2.OVL, OV2A.OVL, and OV2B.OVL. Each overlay file is a separate link, composed of the modules that make it up linked to an origin that is the end of its root module. OV1 and OV2 will be linked to run when loaded at the byte after ROOT.COM; OV2A and OV2B will be linked to run at the byte following OV2. All the link options apply, independently, within the parenthetical specification of an overlay.

The code in an overlay may refer to public labels in its root module, but not vice versa. That's all right; there is an escape. A more serious problem is that LINK80 insists on adding two external references to a root program, "?OVLAY" and "?OVLA0." These presumably mean something to PL/I, though we don't see why the compiler couldn't generate them in the usual way. They have to be supplied because there won't be any output unless all external references are resolved. We resolve them by writing and assembling FAKE.ASM as follows:

```
    CSEG
    PUBLIC ?OVLAY,?OVLA0
?OVLAY:
?OVLA0:
    END
```

Now we can create FIX.OVL and BNK.OVL, the two sections of our BIOS, with all external references resolved and all the code located where we want it. First, FIX.OVL:

```
LINK FAKE &
(FIX=all-12-bios-modules &
    [D4000,LF200] )
```

Because it loads at F200h, none of the bank-BIOS modules will be included in the file FIX.OVL. However, their proper addresses will be filled in where needed. The banked section is trickier:

```
LINK FAKE,all-fixed-modules &
    [LF200,$OZ] &
    (BNK=all-banked-modules &
    [L4000,$OA] )
```

Here we let LINK80 think that there is a root module, loading at F200h, which contains the fixed parts of the BIOS. The "$OZ" switch tells it not to produce any output for this "root" module. Then we tell it to make BNK.OVL, composed of our banked modules and loading at 4000h, the base of our banked BIOS. The "$OA" switch says this file is to be written. All the places in the banked code that refer to fixed code will be filled in with the proper addresses.

And that is how one treads cautiously around the pitfalls and potholes of RMAC and LINK80 to get a two-section BIOS written and linked. The result is two files that contain the code, ready to be loaded into storage in their correct locations and banks. How these parts are put together with the CCP and BDOS of CP/M 2.2 and written onto the system tracks of a diskette, and how during cold start they get loaded and located in storage, is at least two other stories. But arranging the link was the highest hurdle we *had to cross.*

Perhaps that wasn't the most fascinating item you've read in this column. Well, you've nobody to blame but yourself. In the absence of questions and discoveries from readers, our only source of material is the top of our desk. **DDJ**

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

CP/M 83 Show

The recent CP/M 83 exhibition in San Francisco drew over 45,000 attendees, a degree of success surprising even to its sponsors. Browsing through the booths, I was impressed with the very rapid shift towards 16-bit microprocessors that has become evident over the last year. Almost every new machine and application program of any significance was based on an 8086 or 68000, though Olivetti was also present, valiantly trying to rescue the Z8000 from oblivion. There was almost nothing worth mentioning in the way of new 8-bit hardware or software, with one exception: several manufacturers introduced super-miniaturized Z80 CP/M systems in which the whole system board and a slim-line 5" disk drive fit into a chassis smaller than a shoebox.

Digital Research itself displayed many new products at the show, some of which indicate marked shifts in its corporate goals. CP/M-68K was shown running on five different machines including the Sage and Compu-Pro 68000-based microcomputers. It comes with a complete set of software development tools including an assembler and a full-blown C compiler. Reportedly CP/M-68K will eventually be able to execute as a task under UNIX, giving its users the best of both worlds. The same C compiler will be available on the 8086/88 in April 1983. Digital Research also announced that all of its future operating systems products and language compilers will be written in C! Evidently DRI's previous flagship compiler, PL-1 subset G, is being relegated to the wings forever.

Also announced was a Digital Research version of "plain" CP/M-86 for the IBM Personal Computer, including printer spooling and the new graphics support nucleus GSX, for only $60.00. DRI has been discontented with IBM's half-hearted support and marketing of CP/M-86, and is now showing more convincing determination to capture some of the enormous (and affluent) PC user base from Microsoft. This is good news for PC owners, since they can now readily afford both PC-DOS and CP/M-86 and therefore be much more flexible in their application software purchases. But such a drastic drop in price will surely raise questions in the minds of CP/M owners of other machines, who must still pay $150 or more for their operating systems.

The most startling development of all was the Digital Research display of an extended LOGO language for the IBM PC. A seminar on the new product was given by Gary Kildall himself, who described it as having all the benefits of LISP but with a "user-friendly" programming environment. Life is certainly full of surprises.

Intel 8087 News

Hudson Associates has announced an 8087 piggyback board for the Godbout 8085/8088 Dual CPU. This will be a boon to those of you who wish to take advantage of the 8087 for fast floating-point math but don't want to trash your present CPU board for a new $750 Godbout 8086/87 board. The procedure for installing the piggyback board is very simple: remove the 8088 from your present CPU board, plug the piggyback board into the 8088's socket, then plug the 8088 and 8087 chips into the piggyback board. The system should then boot up and run just as before.

This piggyback board will probably also work on other 8088 CPU boards such as the Lomas LPD-88. The main consideration is whether adequate power is delivered to the 8088's socket to supply both processors. The 8087 draws considerably more power (up to 475 mA) than the 8088 (340 mA). We have used a similar hand-wired piggyback board to add an

Command	Description
AR68	Archive utility, stores object files in the C run-time library
AS68	68000 assembler
C	C language compiler
CP68	C language preprocessor for macros
DDT	Interactive 68000 debugger
DIR	Display disk file directory
DIRS	Display directory of "system" files
DUMP	Display contents of a file in hex and ASCII
ED	Line editor
ERA	Erase file(s)
LO68	Linker
NM68	Symbol table display utility
PIP	Transfer, concatenate, and/or filter files between various peripheral devices
RELOC	Relocate a command file to an absolute address
SENDC68	Convert command file to Motorola S-record format
SIZE68	Print the size of a command file

Table 1.
CP/M-68K Commands and Utilities

Offset	Contents
0000-0003	Lowest address of Transient Program Area
0004-0007	1 + highest address of TPA
0008-000B	Starting address of the Text Segment
000C-000F	Length of Text Segment (bytes)
0010-0013	Starting address of the Data Segment
0014-0017	Length of Data Segment
0018-001B	Starting address of the bss (uninitialized data)
001C-001F	Length of bss
0020-0023	Length of free memory after bss
0024-0024	Drive from which the program was loaded
0025-0037	Reserved
0038-005B	2nd parsed FCB from command line
005C-007F	1st parsed FCB from command line
0080-00FF	Command tail and default DMA buffer

Table 2.
Program Base Page Format for CP/M-68K

8087 to the NEC 8086-based Advanced Personal Computer without any problems. Hudson Associates may be contacted at P.O. Box 2957, Santa Clara, CA 95055.

A LINK-80-compatible run-time library for 8087 support has been released by Avant-Code, 1508-A Oxford Street, Berkeley, CA 94709. The library costs $200 and may be used with Fortran-80, Bascom-80, Cobol-80, or Macro-80. Of course, you need to run the linked programs on a system containing the Godbout Dual CPU, since the 8088 must take over control in order to perform the math operations on the 8087.

Preview of CP/M-68K

The information presented here is abstracted from the new CP/M-68K documentation set which was recently released. CP/M-68K is logically symmetrical to the CP/M-80 and CP/M-86 operating systems, with enhancements to support the 16-megabyte memory addressing space of the 68000 microprocessor. The disk file structure is exactly compatible with the 8080 and 8086/88 versions, and supports a maximum of 16 drives with up to 512 megabytes per drive.

The operating system resides in a file named "CPM.SYS" and is loaded into memory by a cold-start routine which is initially read in from the two reserved system tracks (similar to CP/M-86). All of the modules (CCP, BDOS, and BIOS) of CP/M-68K remain resident at all times. The CCP and BDOS are written in C, and the BIOS of course is written in assembly language by the system implementor.

CP/M-68K contains most of the familiar CP/M commands, as well as some impressive new program development tools (see Table 1). The inclusion of a version 7, C-compatible compiler is a tremendous enhancement. Preliminary benchmark results indicate that this compiler generates very efficient code (see Jim Gilbreath's article in the January 1983 *BYTE*).

Transient application programs are designated in the disk directory with the extension "68K." A program may be loaded via a command line at the CCP level or by another program through BDOS funtion 59. After a program is loaded, the Transient Program Area (TPA) contains the base page, the program segments (text, data, and bss). and the user stack (see Table 2 and Figure 1).

The CP/M-68K BDOS functions are very similar to CP/M-86, except that the memory management functions are not included. A typical calling sequence is as follows:

```
move.w #2,D0.W ;move function number
                ;to the first data register
move.w #7,D1.W ;move ASCII bell code
                ;to the second data
                :register
trap #2         ;request BDOS function
                ;to output a character
```

Any results are returned in D0.W. A few of the BDOS functions are different or new compared to the previous operating systems and will be described briefly below.

Function 12 (return version number) yields the value 2022H in register D0.W, signifying a 68000-CPU, single-user environment without networking, and BDOS version 2.2.

Function 50 is a direct BIOS call which allows application software to manipulate the primitive device drivers. The D1.L register contains the address of the BIOS Parameter Block, a five-word memory area containing the desired function number and two other 32-bit paramenters. Any results are again returned in register D0.W.

Funtion 59 loads an executable program file into memory. The address of a Load Parameter Block (LPB) is passed in register D1.L. The LPB describes the program and specifies the load address; it includes the following items:

- address of File Control Block of successfully opened program file
- lowest address of area in which to load program
- highest address+1 of area in which to load program
- address of base page (returned by BDOS)
- default user stack pointer (returned by BDOS)
- loader control flags

The BDOS allocates memory for the desired program and base page and initializes locations 0000-0024H of the latter. Locations 0025-0037H are not initialized. The calling program must fix up any additional program, first pushing a return address on the stack if it wishes to resume execution later.

Funtion 61 (Set Execution) allows a program to link its own handlers to the operating system for such events as bus errors, illegal instructions, and zero divide.

Function 62 sets the calling program into Supervisor State and swaps to the system stack. Presently this call is always successful, though in future versions (especially multi-user systems) the function may not be present or may require certain privilege attributes.

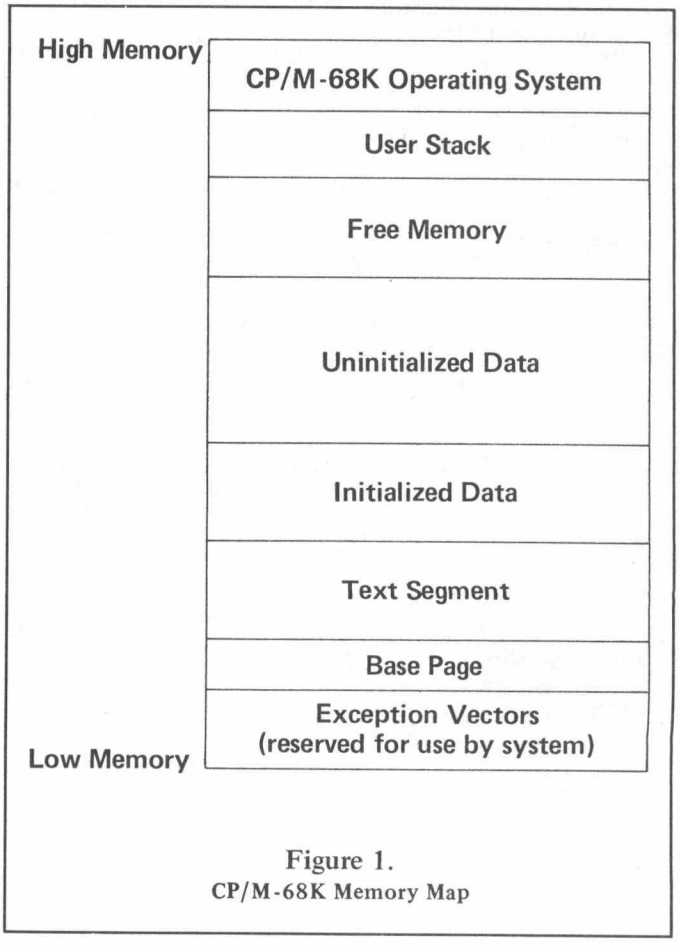


Figure 1.
CP/M-68K Memory Map

In forthcoming columns we will provide examples and benchmarks of programs run under CP/M-68K, and begin to publish some 68000 utility subroutine listings.

Improved Square Root Routines

Don Taylor of Corvallis, Oregon writes: "If you are going to do square roots, and you have assembly language access, don't use Newton's Method. It is a fine tool, and very general purpose, but too slow for something as simple as this. I don't remember where I first ran across the integer bit-by-bit method, but typically it is an order of magnitude faster.

"The idea behind the enclosed square root routine is very similar to the way that it is done by hand, except for the fact that the process is even simpler in base 2. In decimal you use pairs of digits, in binary you use pairs of bits. In decimal you must guess the next trial digit, in binary it can only be zero or one.

"If you breakpoint the routine at the bottom of the loop, you can see the partial result growing bit-by-bit. In addition, the remainder should not exceed the trial root. If the remainder exceeds half the root at the end of the routine, the root can be better approximated by increasing it by one; without doing this the root will always be less than or equal to the real square root."

Our thanks to Don for this interesting and elegant subroutine. A listing in Intel 8086 mnemonics accompanies this column on page 82.

DDJ

Integer Square Root

(Text begins on page 79)

```
                                        title   'Integer Square Root'
                                        pagewidth 79
                                ;
                                ; Integer Square Root for 8086/88
                                ; Contributed by Don Taylor
                                ;
                                ; Call with:      AX = Argument
                                ; Returns:        AX = Square root
                                ;                 Other registers preserved
                                ;
0000 53                         sqrt:   push    bx          ;save registers
0001 51                                 push    cx
0002 52                                 push    dx
0003 8BD0                               mov     dx,ax       ;argument into DX
0005 B90800                             mov     cx,8        ;number of iterations
0008 33DB                               xor     bx,bx       ;clear the remainder
000A 8BC3                               mov     ax,bx       ;clear trial value and
                                                            ;final result store
000C D1E3                       sqrt1:  shl     bx,1        ;double partial result
000E 43                                 inc     bx          ;guess next bit is a 1
000F D1E2                               shl     dx,1        ;fetch 2 new bits
0011 D1D0                               rcl     ax,1        ;from argument
0013 D1E2                               shl     dx,1
0015 D1D0                               rcl     ax,1
0017 2BC3                               sub     ax,bx       ;do a trial subtraction
0019 7308               0023            jnc     sqrt2       ;guess was right,
                                                            ;append a 1 bit
001B 03C3                               add     ax,bx       ;guess wrong, put it
                                                            ;back
001D 4B                                 dec     bx          ;and clean up for
                                                            ;next pass
001E E2EC               000C            loop    sqrt1
0020 E90300             0026            jmp     sqrt3       ;go scale result
0023 43                         sqrt2:  inc     bx          ;convert xxxx01 to
                                                            ;xxxx10, i.e. append
                                                            ;a 1 bit
0024 E2E6               000C            loop    sqrt1
0026 D1FB                       sqrt3:  sar     bx,1        ;divide by 2 to get
                                                            ;actual square root
0028 8BC3                               mov     ax,bx       ;return result in AX
002A 5A                                 pop     dx          ;restore other registers
002B 59                                 pop     cx
002C 5B                                 pop     bx
002D C3                                 ret
                                ;

END OF ASSEMBLY.  NUMBER OF ERRORS:    0.  USE FACTOR:  0%

A>
```

End Listing

SOFTWARE REVIEW

Unica and XM-80

by Michael Favitta

Product: Unica and XM-80
From: Knowlogy
P.O.Box 283 – E
Wilsonville, OR 97070
$195.00 ($95.00 Unica only)
Reviewed by Michael Favitta

Knowlogy advertises that its software package "brings a Unix-like environment" to any Z80-based computer system running CP/M version 2 or beyond. The Unix environment is simulated by using a set of utility programs that possess Unix-like attributes. If you purchase XM-80, you will also need Microsoft's MACRO-80 assembler and LINK-80 loader.

The package is divided into two independent parts, the executable modules called Unica and the language translator XM-80. The Unica are a group of file-related utility programs that perform functions such as delete, copy and search. They are written in the XM-80 language and the source for each is provided. The XM-80 translator converts the XM-80 source modules into standard Z80 assembly language. The Unica may be purchased without XM-80 for $95. If desired, XM-80 may then be added at a later date for the cost difference plus a $10 handling charge. You may also purchase new and enhanced Unica as they become available but Knowlogy does not indicate what the fee for this service will be.

Unica

The Unica part of the package consists of eighteen COM files, each of which performs one function and is referred to as a command. To create the Unix-like environment, all commands support redirection of standard I/O to any device or disk file, multiple commands linked together via "pipes," and filename conventions that support user numbers and extended wild card capabilities.

The commands provided are:

BC	Binary file comparator that displays differences between two files.
CAT	Concatenates one or more files.
CP	Copies one or more files.
DM	Prints a disk allocation map.
FID	Computes a 16-bit checksum for a file.
HC	Concatenates one or more files horizontally (by line).
LN	Creates an alias for a file via a directory link.
LS	Lists the directory contents.
MV	Renames (moves) one or more files.
RM	Deletes (removes) one or more files.
SC	Source file comparator that displays differences between two files.
SFA	Sets and changes file attributes.
SP	Spelling error detector.
SR	Pattern searching with wild cards.
SRT	Sorts lines of a file by text in specified columns.
TEE	A "pipe fitting" used to direct output to more than one command.
WC	Counts the characters, words, and lines in one or more files.
WX	Extracts words from one or more files.

Each command has a set of flags associated with it that are used to specify procession options. Flags consist of a minus sign followed by one or more letters. They may appear separately or in groups, any place in the command line. Two of the most useful flags are the N and V flags. The N flag prints the name of each file as it is processed. The V flag takes this a step further and allows you to verify that processing should be carried out on each file specified. This allows wild card filenames to be used freely, as you can prevent any file from being processed even though it matches the wild card. This feature can prevent some costly mistakes when using a command such as RM to delete files.

Many of the commands are more useful than the brief descriptions of them above may indicate. The commands CP, HC, and SRT are such commands and merit further discussion. The spelling error detector (SP) also needs some explanation as it was the only Unicum that performed poorly.

The copy command (CP) has all the capabilities of PIP except for the data conversion functions (such as converting the output to upper case). In addition, CP allows you to set or change file attributes and will not overlay a file that is marked as write-locked. It can copy a file to the same disk so you can make backup copies easily on a single-disk system. The command is designed for maximum disk efficiency as it uses all available memory for buffers and always reads as much as possible before writing. In my tests, I found that CP was about twenty percent faster than PIP when two or more files were being copied. When copying one file they were about the same speed. As with all file-related commands, extended wild card capabilities are present that can be controlled with the V flag.

The horizontal concatenate command (HC) does a lot more than just append lines together. If you specify different sets of flags, HC will expand tabs to spaces, compress spaces to tabs, add leading spaces, remove trailing spaces, print one or more files side by side starting at any column position, and append a string constant to each line of a file. It is a handy tool for reformatting text to fit various terminal screen widths and for converting files from other systems to your own space/tab conventions.

The data sorting utility (SRT) sorts the contents of a file into alphabetical order. The size of the file that can be sorted is determined by your system's memory size, as sort is done in memory using the Quicksort algorithm. A primary sort key field can be specified by using flags to set the column positions. In this case the rest of the line will be used as a secondary sort key.

The spelling error detector (SP) creates a list of all the words in a file that it does not find a match for in its dictionary of 23,688 words. Unfortunately, the dictionary does not include plurals or conjugated verbs and there is no facility to add new words. These limitations are documented and Knowlogy promises to upgrade this command in the future. In actual testing, a document that contained 773 words had 680 of them flagged as possible errors. Only 74 of the words flagged were actually incorrect (e.g., misspelled, acronyms). Not wanting to believe the results, I created a file that contained 35 common words and ran SP again. To my disappointment, it flagged 28 of the words as possible spelling errors. This was also the only Unicum that I found any software problems with. SP causes a system crash if the dictionary file is not present when it is executed. In its current form, the SP Unicum is not worth using.

The Unica package is more than just a set of independent utility programs. The commands may be linked together to form complex procedures. These connections are called pipes. Along with the ability to redirect the output of a command to any device or file, pipes create a flex-

ibility that makes the Unica package a worthwhile investment. The following examples demonstrate the command line syntax, the redirection of output, and the use of pipes.

```
CAT file1 file2 |
   SR "find this" -1 >b:file3.ext;2
```

This command string concatenates file1 and file2 in a temporary file that is passed to SR. SR writes the line number and line to file3.ext on drive b for each line that contains the string "find this." File3.ext is associated with user number 2.

```
WX -v *.doc | TEE con: | SRT -u |
   TEE lst: > file1
```

This command string extracts a list of all words used in all files that have a .doc extension on the currently logged drive. You will be prompted to verify that you really want to process each file. The extracted words are printed on the console and piped to the sort (SRT) command to sort the list of words and remove all duplicates. The sorted list is written to the printer and file1.

With the exception of SP, I found the Unica package to be well though out with respect to what commands were provided and how they interacted when connected via pipes. Each command worked exactly as documented. Not all the commands worked perfectly in every case but the cases that caused problems were clearly defined in a "bugs" section for each command. Besides SP causing a system crash when the dictionary was not present, the only problem I encountered had to do with the physical size of the package. I have single-density 8-inch disk drives that have a formatted capacity to 250 Kbytes. The Unica without the spelling dictionary (another 73 Kbytes) take up 222 Kbytes of storage. This leaves very little room on a single disk for other utilities and text or source files. Since all temporary files that are created during processing (via pipes) reside on the currently logged disk, the problem is compounded. This problem can be overcome by using more than one drive. You then must specify the drive for all the files and commands that do not reside on the currently logged drive. This procedure can be quite frustrating as there is a much greater chance for human error, especially when entering complicated command sequences. Dual-density disk drives significantly enhance the practical usability of this product.

XM-80

XM-80 is a programming language that stresses the design of programs into modules that can be used in more than one program. The analogy of an electrical engineer, selecting existing integrated circuits and wiring them together to perform a required function versus using all discrete components, is used to describe this approach. Knowlogy calls this methodology "software synthesis." In keeping with the analogy, each module is called a component and related components are grouped into families. Two families of components are supplied with XM-80. The 2800 family consists of components for the Z80 that are not hardware or software dependent. The 2801 family are components that require CP/M version 2 or beyond. All the Unica are built from the components in these two families, "wired" together with Z80 assembly language. The source code for the two component families is not provided. Knowlogy's software licensing agreement does allow you to freely distribute any software that you develop, even if it contains one or more of the components. The agreement limits the distribution of any programs derived from a Unicum to other license holders.

XM-80 is an acronym for an extension of Microsoft's 8080/Z80 assembler, MACRO-80. The XM-80 compiler translates XM-80 procedures and/or procedure calls into standard Z80 assembly language that can be assembled by MACRO-80. Any line that is not recognized as an XM-80 statement is passed through unchanged, allowing any mixture of assembly language and XM-80 to be used in writing XM-80 programs. Using XM-80 forces programmers to abide by a rigidly defined set of interface rules when writing new components. These include component naming and documentation conventions. By forcing adherence to the component interface rules for each component, the XM-80 compiler can validate that the arguments passed and the registers used by the specified procedure are correct. The end result of using a tool such as XM-80 is to increase programmer productivity and decrease program development time by not reinventing the wheel for each new application. The compiler also reduces the possibility of program errors caused by passing parameters to subroutines incorrectly.

The following is a summary of XM-80 programming guidelines and syntax.

The rules for XM-80 program development are:

(1) Search through the available components and select those which are applicable to the project.

(2) Define any new components needed. Code, test, and document each new component. Enter the completed component and documentation into the correct family library.

(3) "Wire" together the components with assembly language to produce the completed program.

The interface rules for components are:

(1) Component names consist of up to six upper-case letters to identify the component, followed by a four-digit family name, followd by two lower-case letters to denote the author. If more than one global symbol exists for a component, then three letters are used to identify the component and global symbols are created by appending up to three letters to the component name.

(2) The syntax for procedure definition and procedure calling is demonstrated by the examples below.

```
proc GETNUM [CHAN:stk] →
   [ERROR:a,NUM:hl] +C-de
begin
Source code (can include calls to
   other components)
end GETNUM
```

The procedure GETNUM is defined as having one input parameter that is passed on the stack. An error code is returned in the accumulator and a number in the HL register pair. The carry flag is altered and register pair DE is destroyed.

The GETNUM procedure is called by:

```
GETNUM [stk=(VALUE)] →
   [a,SAVEHL=hl] +C-de
```

The contents of the 16-bit word pointed to by VALUE are pushed onto the stack and then the procedure GETNUM is called. The contents of register pair HL are stored at SAVEHL after the return.

The procedure definition is also entered in a special file called the component interface file (CIF). The CIF is used for parameter error checking by the XM-80 compiler. Note that the registers destroyed by the called routine must be listed on the procedure call. These are checked by the compiler to make sure that the programmer is aware of all registers that will be affected by the procedure call.

The XM-80 compiler understands several statements besides those used in procedure calling and definition. These statements are used to give the programmer control over how XM-80 sets up procedure linkages and where needed CIF files can be found. The compiler can also be instructed to generate inline code if desired. The compiler itself is not a Unicum so it does not support any of the Unica capabilities such as redirection of output. It uses a command syntax similar to that of MACRO-80. Most of the command line is used to generate a MACRO-80 call after XM-80 is done. The XM-80 compiler is controlled by specifying a set of flags similar to those used with the Unica at the end of the command line.

Again, the package worked as documented although the "known bugs" section documented some serious problems (note that the version I reviewed was released in March 1981). The worst of these is that if you enter the extension on the source input file, it is deleted. I would

have liked to have seen a warning at the very front of the manual about this problem, as it is an easy mistake to make and the consequence could be hours of lost work. The XM-80 package is a worthwhile assembly language programming development tool. It includes an extensive set of library components that could save many times the price of the package in programmer time. An added bonus is that by using the I/O components provided, your programs can contain all the features that the Unica possess, including the ability to build pipes.

Unica and XM-80 Documentation

The documentation for Unica and XM-80 comes in a three-ring notebook with labeled dividers that make it simple to find a desired section. This format makes it easy to add to and update the manual as you develop new Unica or XM-80 components. My one big gripe with the manual is that it doesn't have a usable page-numbering scheme. All Unica and components start at page 1. This makes an index useless, so Knowlogy did not include one. There isn't a table of contents or any kind of summary of which Unica and components come with the package either. This means you have to read the entire manual to get a feel for what the package offers. The manual becomes tedious to use once you've become familiar with the package and just want to use the manual as a reference. This problem could be solved easily by the addition of a couple of appendices with summaries of the Unica and XM-80 components and their associated syntax and argument lists.

Putting format aside, the manual is written carefully and thoroughly describes the product and how to use it. Examples are used liberally throughout the text to demonstrate what the various Unica do and how they can be combined to perform other functions. For XM-80, an example is provided that takes you from program design, through testing, to program completion. Detailed data sheets are provided for each of the components. The source codes for the Unica can also be used as an excellent tutorial on how to write XM-80 programs, as they are well written and extensively commented.

Knowlogy dispenses a good deal of software design philosophy along with the hows and whys of Unica and XM-80. This makes the manual read more like a textbook than a reference manual. (This probably has something to do with why they chose "Knowlogy" as a company name.) The entire package is presented as a complete methodology on how to solve problems with a computer, with the Unica part of the package an end result of the methodology. Personally, I like this approach and think the resulting manual is one of the best that I have seen with respect to content.

Summary

The Unica are a group of utility programs that use an extended CP/M command sequence that allows the output of one command to be used as the input to another. With the exception of the spelling command (SP), all the Unica worked as documented and would make a useful addition to just about anyone's set of utility programs. Even without SP, I think the Unica package is well worth $95.

XM-80 is a software development tool designed to increase programmer productivity. If the "software synthesis" methodology is followed, then this tool is a worthwhile investment. If you can't live within the rigid framework defined by the methodology, then I would suggest saving the extra $100 that XM-80 costs. To get the most out of the entire package, you should design your programs with the existing components that provide the special features used by the Unica.

Overall, the Unica/XM-80 software product is designed and documented very well. The package I received did have a few problems, which I presume will be corrected in later releases, but most are documented and can be programmed around.

We checked with Knowlogy to see if there had been upgrades from the release that our reviewer received. We were informed that they have indeed released an upgraded version that has several improvements, among them a redone spelling error detector which now has a dictionary of 130,000 words (including plurals and conjugations) and the ability to add new words. – Ed.

OF INTEREST

by Michael Wiesenberg

Baudy Modem

Modems with 1200 baud are lowering in price. **Racal-Vadic's VA212LC** costs $550. The Bell-compatible, switched-network, originate/answer full-duplex modem operates at 1200, or 0 to 300 for communications with slower systems, automatically determining the baud rate of a calling modem. Auto-answer can be disabled to obviate answering calls on lines that carry both voice and data. Five LEDs display operational status, monitoring transmit and receive data, carrier detect, voice/data status, and bit rate; and the unit continually monitors itself with automatic self-test routines. The VA212LC automatically handles 9- or 10-bit character codes, and thus transmits EBCDIC or ASCII. **Reader Service No. 101.**

A Genius for Displays

Many of the newer dedicated word processing systems and computers have full-page screens, but now you can get an 80-character x 57-line display (73 optional) for Apple and most other computers with standard RS232 port. **The Genius** from **Micro Display Systems, Inc.**, offers white, green, or amber phosphor, reverse and flashing video, 128-character ASCII set (or optional foreign character sets), graphics, 19.2K baud interface, and internal 16K memory buffering and screen memory, on an 8- x 10.5-inch display. You can get 100/120 volts, 60 Hz for $1795, or shielded 220/240 volts, 50 Hz for $1950. (Prices are FOB Minneapolis, which, I think, means you have to pay for shipping from there to anywhere else.) **Reader Service No. 103.**

And Another Big Monitor

QuadScreen, from **Quadram**, for the IBM PC is a 17-inch monochromatic display with 160 characters x 64 lines, more than five times the screen capacity of IBM's monitor. Bit-mapped graphics permit individual addressing of all 960 x 512 dots, while the hardware can display any character font in any size (providing infinite user definable character sets). QuadScreen has its own character set in a 5 x 7 matrix, while software provided by Quadram permits defining characters of any size or shape. Reverse video and smooth forward and backward scroll are part of the product. $1950 gets you the monitor, cable, software, and display controller board that fits into any expansion slot of the PC and has 64K memory that can be used alone for memory expansion.

Quadram also makes **QuadSpooler**, a software program that permits simultaneous printing and spooling, even with two printers. You designate from 2K to 56K of RAM to use as the spooler buffer. No additional hardware is required, though Quadram would like to sell you one of their Quadboard memory expansion boards for the PC. Buy one and get QuadSpooler free, or get just the software for $19.95.

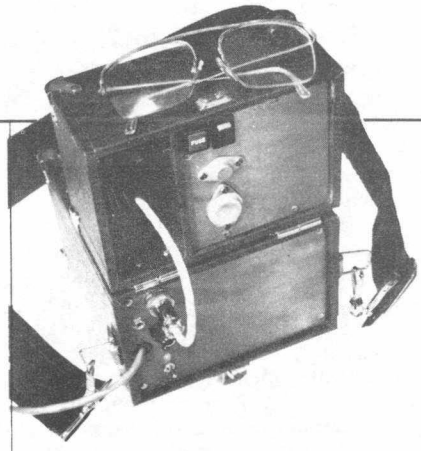
And while I'm giving them space, Quadram's **MicroFazer** has improved since last I mentioned it. This hardware buffer/spooler has a copy feature that prints as many additional copies of the buffered information as specified by pressing a button. Use it with any computer or printer, and expand its 8K memory size up to 512K (without using any of the computer's memory). Parallel-to-parallel, parallel-to-serial, serial-to-parallel, and serial-to-serial are all available, priced from $169 to $1395. **Reader Service No. 105.**

And An Inexpensive Amber Guy

The **ADP-120A** from **Atlantic Data Products**, a 12-inch, high-resolution, amber phosphor (amber on grey), 16MHz-bandwidth data monitor, compatible with many micros, costs $169. **Reader Service No. 107.**

Rufus T?

The **Firefly** battery pack (and eyeglass stand) from **Gamma Research** gives three to eight hours of portable battery power to Osborne, Apple, IBM, KayPro, Franklin Ace, et al. An optional uninterruptible power supply (U.P.S.) protects micros from voltage spikes or blackouts by automatically switching to the Firefly whenever sensing AC failure and functions as a rapid (three- to four-hour) Firefly



charger. You get a 12-volt DC power cable to run the computer or charge the Firefly from your car's cigarette lighter, four LED charge state monitors, a discharge warning buzzer, and a 12-month warranty. It all comes in a detachable (into two sections) 6½- x 4½- x 9½-inch hard travel case with a carrying strap. The Firefly battery pack reduces computer heat generation by up to 80%, they say, which extends the computer's life. The battery pack is $229 to $279, and U.P.S. $99 to $109. And to double battery power time, they have a booster pack for $169. Gamma Research also offers a Firefly 9-inch, 1000-line, high-resolution, green ($244), or amber ($299) monitor in a lightweight travel case with a detachable daylight glare-reducing hood, that runs off AC, your car's cigarette lighter, or (or course) the Firefly battery pack. **Reader Service No. 109.**

Basic Database System for CBASIC and Other Bases

The **Tarbell Database System** from **Elliam Associates** is supplied in .COM files *and* CB80 source (to run under CBASIC). It supports up to 19 open files, with no limit on numbers of records or their lengths or those of field names. The system's series of programs uses a common file format for random and sequential files, for optional index files, and to chain from the main menu to other programs and HELP files, set up files, enter data, update files, write reports, build command files, copy files from one format structure to another, sort files, print mailing labels, and personalize letters. A QUERY language interactively or from command files defines search area, scope of search, and search conditions. You need a 48K sys-

tem. $100 for disk in most configurations and formats (specify), plus $1.50 p & h (Californians, add sales tax). **Reader Service No. 115.**

Yeah, But Where's Ferddy?

MORTTY is a communications program from **Phillip Emerson** for H8, H89, Z89, and Z90 computers. It sends and receives ASCII and Baudot code at any baud rate the equipment can handle, uses a full screen and a user-definable split screen, has a type-ahead buffer, logs on automatically, allows the user to define handling of carriage return, line feed, and control characters, echoes or not as defined, uses full or half duplex, automatically identifies Morse code, has advanced string storage, automatically programs smart modems, printers, etc., and has user-definable, single-keystroke functions. You need HDOS with 32K and a modem. You get a 5¼-inch hard-sectored disk, a 60-page manual, and a year's free updates, for $100 (Ohio, add sales tax). **Reader Service No. 117.**

Alien Contest

The **Alien Group**, who makes Voice Box speech synthesizers for Atari and Apple computers, would like to give away $6,800 to the authors of what they judge to be the best talking or singing programs that use (what else?) the Voice Box, in their **Voice Box-ing Match Contest** (cute, cute). First prize is $5,000. Judges will be computer game players ages 13 to 18. The Alien Group also wants you to know about their new Atari Voice Box that recognizes phonetic or normal spelling, has an (apparently) unlimited vocabulary and random sentence generator, is programmable from BASIC or assembly language, plugs directly into the serial port, does not blank the screen while talking, is available in 16K cassette and 16K and 32K disk versions, speaks with intonation or feeling (but they don't say if it can do *both* simultaneously), sings in key (that's better than I can do), sings in three simultaneous musical tones, converts two keyboard rows into a literal chromatic piano with a 2½-octave range, programmatically handles glissando, tremolo, and vibrato, comes with 40 songs that you can add to, has a system for easily (so they say) creating and editing words and music, has educational software, has two "talking heads" with programmable lip-sync animation and high-res color, and inclues a demo. $169 for a wonderful product that the *Las Vegas Sun* thought was the highlight of COMDEX. **Reader Service No. 123.**

Process Words for Less

Word processing software can cost upwards of $500 and may not be worth the price, or cost under $100 and not work worth beans. Here's one from a company known for quality software (in the games field mainly, but they do have a reputation to maintain) that seems to offer a lot for $69.95. **Bank Street Writer** from **Brøderbund Software** ("Developed," they say, "and heavily tested among students and young adults by Bank Street College and Intentional Educations" – is that the name of a *real* school?), for 48K Apple II+ or Apple II with Applesoft in ROM or RAM and 16-sector controller and for 48K Atari 400/800 (and maybe 1200?) (requiring BASIC cartridge for tutorial only), has global search and replace, move and unmove, automatic centering and indent, inverse text highlighting, word wrap, disk storage and retrieval functions with password protection (you don't get *that* on most WPs!), redefinable defaults, and a "potent" print format routine, including document chaining, page headers, page numbering at top or bottom, partial printing, and format previewing before printing. No special hardware is needed and upper and lower case are displayed. The back of the disk has a tutorial, and you get a free backup disk and complete documentation (I think that means a manual). **Reader Service No. 119.**

"-trix" Are Not Necessarily for Kids

Quotrix, the second of **Insoft**'s "-trix" educational games for *the IBM* PC has players guess one of 700 possible famous quotes by following a trail of crossword puzzle clues, trivia questions, foreign words, and other word games. Sounds like a lot of fun for $34.95. **Reader Service No. 121.**

ET Enthusiasts

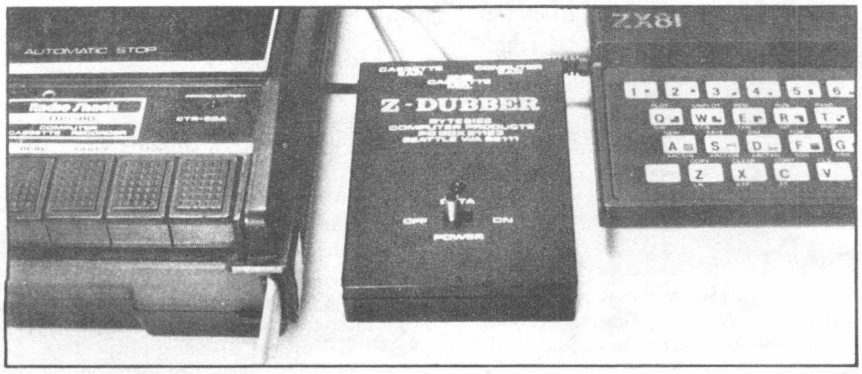
Join **ETUG**, the user's group for Heath's ET/ETA-3400 microprocessor trainer (but not affiliated with Heath) and for those who program the 6800 and 6809 and in tiny BASIC, and receive a quarterly newsletter for $16 a year (US and Canada; $22 elsewhere). They've been around for over a year. **LAETUG** for Los Angelenos has just fired up, and meets at a Heathkit Center. ETUG, 11231 Oak St., El Monte, CA 91731; LAETUG, c/o Gilbert Murillo, Heathkit Electronic Center, 2309 S. Flower, Los Angeles, CA 90007; (213) 749-0261 and 443-2237.

Logo For Atari

Logo for the Atari 400, 800, and 1200XL, developed and manufactured for Atari by Logo Computer Systems, Inc., of Montreal, will soon be available in a 16K cartridge sold and distributed by **Atari** for under $100. **Reader Service No. 113.**

Learn Modula-2

From the Father of Pascal, Nicklaus Wirth, came Modula-2 (*MODU*lar *LA*nguage) in 1980, sometimes called "Pascal for grownups," and described in understated fashion by Wirth himself this way: "The structure of Modula is an improvement upon the structure of Pascal." Yes, there was a Modula-1, the developmental version, in 1975.



Load Bettair Weeth Z-Dubber

Improve on the current difficulty of loading cassette programs into the Sinclair ZX81/Timex 1000 with **Z-Dubber** from **Bytesize Computer Products** or connect two cassette recorders for making perfect backups. $29.95 plus 90¢ shipping. **Reader Service No. 111.**

Logitech is developing Modula-2, together with a fully symbolic debugger, for various 8086 CP/M-86 and MP/M-86 environments, and would be pleased if you would contact them for more information. **Reader Service No. 129.**

And to learn all about the language, **Volition Systems** are selling a 264-page loose-leaf format **Modula-2 User's Manual**, accompanied by Wirth's 48-page Modula-2 technical report, for $35. It contains information about standard library modules, utility library, and implementation on UCSD Pascal, and a machine-specific implementation guide with information on libraries, interrupt handling, and machine-level data representation. You can also buy Wirth's Springer-Verlag book, **Programming in Modula-2**, from them for $16. **Reader Service No. 131.**

Tick TOC

The **Technology Opportunity Conference** (TOC), a worldwide convergence of optical storage, videodisk, and computer technology sponsored by *OPTICAL MEMORY NEWSLETTER Including Interactive Videodisks* and Office of the Future Limited (heady names indeed!), will meet in San Francisco, April 5-7; New York, April 26-28; Washington, June 14-16; London, July 5-7; Los Angeles, July 11-15; New York, September 12-16; Chicago, October 10-14; San Francisco, November 8-10; Kaanapali (Maui), November 29-December 1; Orlando, December 5-9; and Houston, January 9-13. **Reader Service No. 127.**

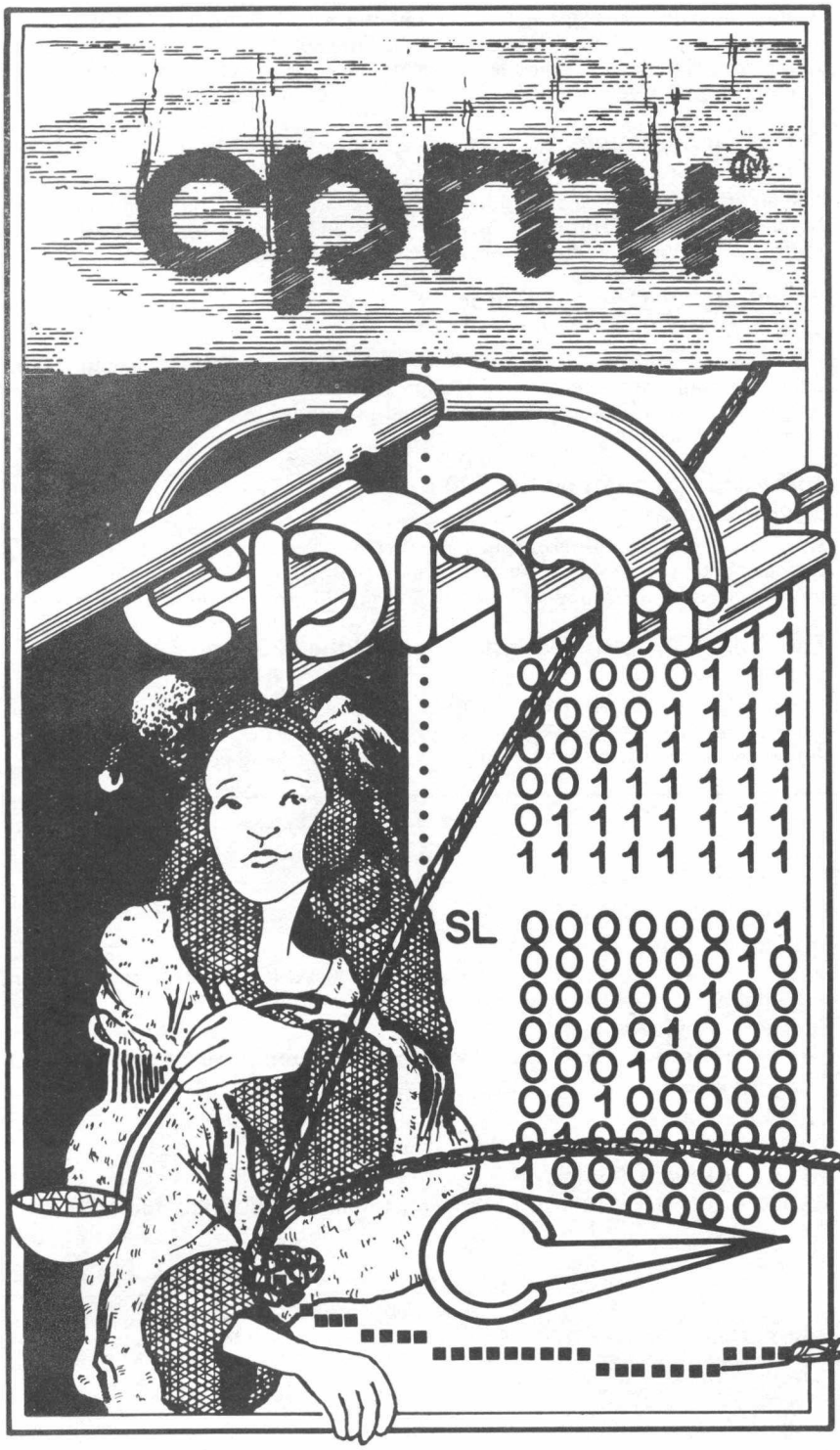
Can Aerobics for the Computer Professional Be Far Behind?

I've got my copy of **Tone Up at the Terminals**, "an exercise guide for workers in automated offices." You can get yours too, *free*, from **Verbatim Corporation**. Now, without leaving the confines of my terminal, I can trim my hips and waistline by doing the Windmill, release tension in my hand and wrist with the Wrist Flex, do the Knee Kiss, and, to firm and tone my legs and (blush) buttocks, I can do the Derriere Firmer. The 12-page booklet includes over 20 exercises that have been reviewed and approved by the California Governor's Council on Wellness and Physical Fitness. **Reader Service No. 133.**

79 May 1983 $2.50 ($3.00 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems



AUGUSTA Part III

A Fast Circle Routine

Shifts and Rotations

SBC, TSX and TXS

Plus:
Small-C Screen Editor Enhancements

Dr. Dobb's Journal

For Users of Small Computer Systems

May 1983 Volume 8, Issue 5

Publisher – Clifford West
Editor – Reynold Wiggins
Managing Editor – Craig LaGrow
Editorial Consultant – Marlin Ouverson
Contributing Editors –
Robert Blum, Dave Caulkins, Dave Cortesi, Ray Duncan, Michael Wiesenberg
Marketing Director – Craig Harper
Marketing Manager – Beatrice Blatteis
Advertising Director – Carl Landau
Advertising Sales – Doug Millison
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Terri Marty, Linda Marohn
Production Manager – Barbara Ruzgerian
Production Assistants –
Jane A. McKean, Alana Hunter
Typesetter – Paula Fairchild
Cover Illustration – Al McCahon

Copyright © 1983 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: DeWitt S. Brown, Burks A. Smith, Robert M. Connors, Robert C. Luckey, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Thomas Davis, Jan Steinman, G. Hunter, Ronald E. Johnson, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III, Infoworld, Stan Veit, Western Material Control, S.P. Kennedy, Ed Moran, W.D. Rausch. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

13 Augusta Part III – The Augusta Compiler
by Edward Mitchell

Having discussed language definition and the p-code interpreter of Augusta in previous issues (*DDJ* Nos. 75 and 77), Mr. Mitchell this month begins discussion of the recursive-descent compiler of his unofficial Ada subset. The listing of the Augusta compiler begins in this issue and continues in the July issue.

32 A Fast Circle Routine
by Daniel L. Lee

The algorithm presented here draws complete circles accurately and quickly, and is relatively easy to program. Written in assembly code for the IBM PC, the routine can be implemented as an IBM Pascal procedure.

38 Enhancing the C Screen Editor
by Alan D. Howard

In the January 1982 issue, Edward Ream published a Small-C screen editor. Since then, its popularity has continued to increase. Mr. Howard provides enhancements to the utility that should prove useful and instructive to those wishing to expand the editor's capability.

64 Shifts and Rotations on the Z80
by Ron Goodman

Shifts and rotations are useful in a number of programming contexts. For those curious about this group of instructions on the Z80, but unable to find a satisfactory description, Mr. Goodman's discussion should prove enlightening.

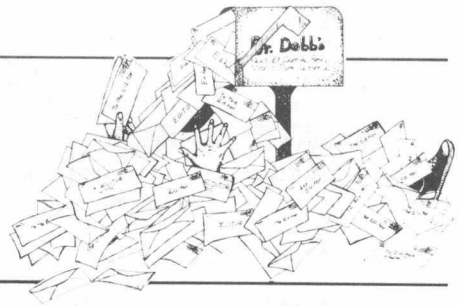
67 The SBC, TSX, and TXS Instructions of the 6502 and 6800
by B.T.G. Tan

As one might expect, the 6502 and 6800 CPUs have a number of common instructions. Some of those, however, have subtle differences of which the programmer should be aware. Three such instructions are described and compared.

7 Letters
7 Editorial
10 Dr. Dobb's Clinic
More on disks, and Z80 hang-ups; Controlling MBASIC; The Buffered Keyboard; items on CCS, CompuPro, and more.
69 Software Reviews
ZAS Z8000 Software Development Package; 6809 Cross Assembler.
70 Book Reviews
74 CP/M Exchange
The first in a series of columns taking a look at CP/M *Plus*.
80 16-Bit Software Toolbox
MSDOS vs. CP/M-86; IBM PC Character Set Linker; FLIP Utility for the IBM PC.
88 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

LETTERS



Likes to C It Standard, Too

Dear Dr. Dobb,

I had to jump up and cheer when I read Richard Foulk's letter (Standard Deviation) in *DDJ* No. 77. I am a relative newcomer to C, so when I started seeing deviations from Kernighan & Ritchie, I was dismayed but figured maybe they knew what they were doing. I am glad to see that someone else feels the same way.

I bought Software Toolworks's C/80 which, though not the full implementation, does not deviate from K&R. My hat is off to Walt Bilofsky and his crew. Then I started transferring C User Group software from CP/M to HDSO (Heath) media so I could use it. The hardest part was translating the file-handling functions to the K&R standard to which C/80 adheres. (Are you listening, Leor Zolman? Please bring BDS C back to the K&R fold.)

I always thought that one of the greatest advantages (among several) of C over Pascal was the greater uniformity and portability among versions. Now, that seems to be fading away.

Meanwhile, take a pat on the back for putting out a fine journal, and keep those C features coming.

Sincerely,
Robert L. McClure, Jr.
Route 1, Box 158
Troy, IL 62294

CREF Update

Dear People:

Several people have called with questions about "Cross-Reference Generator in C," in *DDJ* No. 68. The code fragment at the end of line 53 of "cref.c" is actually the end of line 99. The Whitesmith's implementation of alloc() takes two parameters. The first is the number of bytes of space to allocate and the second is put in the first word of the allocated space. All linked lists in the program use the first word in the structure as a link. The whole scheme works very nicely for linked lists implemented this way.

The recursive implementations of lookup() and tree_walk() are elegant, but can use large amounts of stack space. With large files on CP/M-80, this can cause the stack to overwrite the space allocated from the heap. Revised listings are included with this letter (see Figure 1, page 9). Although the revised tree_walk() occupies more space, the capacity of the revised program is larger. Algorithm T of *The Art of Computer Programming*, Volume 1 is the basis of the revised tree_walk(). The stack used in Knuth's version is implemented within the tree being traversed. The left link field is used as a pointer to the node under it on the stack. The tree is destroyed as it is traversed.

One additional trick: when more files are being cross referenced than will fit on a command line, use #include statements on the standard input.

Thank you to those people who called or wrote.

Sincerely,
Jeff Taylor
The Toolsmith
139 G Street, No. 151
Davis, CA 95616

More Benchmarks for the PC

Dear Sir:

We have performed a benchmark test using Fortran and BASIC on several computers. The program was a simulation of the dynamics of a "floating-ring seal" used in high-speed turbomachinery. This is a typical scientific/engineering application, which involves lots of number-crunching. The motivation here was to assess the speed of the IBM PC when used for engineering or scientific applications, and to compare it with other personal and mainframe computers. All tests were performed in single-precision. The results are shown in Figure 2 on page 9 (numbers in parentheses indicate timing with Intel 8087 Math Coprocessor and appropriate libraries).

In a further test, we have run the

EDITORIAL

This issue we welcome two new faces to our ranks. Bob Blum takes over the CP/M Exchange column this month and presents his first installment of a series on CP/M Plus. He will be exploring the features of this new product over the next few months, as well as looking at how to get more from the CP/M 2.2 that you may already have and addressing the usual mailbag items. His knowledge of the ins and outs of CP/M should prove helpful to all of us.

The other new face is Craig LaGrow, our managing editor. Craig is a science writer finishing his master's degree in journalism at Stanford University. With a strong background in publishing and keen interest in computer technology, he will ensure a continued increase in the quality of our publication.

* * *

Many of you have discovered the "quick input" card that appears as an insert with the reader service card. For those of you who have not, let me point it out as an excellent way to drop us a quick note. There is no substitute for a full letter that can be shared, when appropriate, with the readership at large; but this card is a fast, easy way to send suggestions or comments. Either way, we love hearing from you.

* * *

For those who have been wondering, we *are* planning our annual Forth issue. We think you will find it interesting, as always. Further down the road, we would like to publish another telecommunications issue. We have no definite date set, but are looking toward late 1983. Those interested in contributing to an issue on this timely topic should get in touch with us.

On the subject of general contributions, we would like to encourage articles on a wide range of topics. A few that come to mind immediately include the 68xx series of CPUs, 16-bit utilities and applications, C and Unix. If you have an idea or article in an area that might interest our readers, please contact us. We will be glad to discuss it with you.

– Reynold Wiggins,
Editor

Figure 1.

```
/* lookup - find pointer to node for "name" */
struct node **lookup(root, name)
  register struct node **root;
  register char *name;
  {
  register int c;

  while(*root && (c = lexcmp(name, (*root)->name, 0)))
    root = c < 0 ? &(*root)->left : &(*root)->right;
  return root;
  }

/* tree_walk - walk a binary tree, doing ftn to  ach node */
tree_walk(root, ftn)
  register struct node *root;
  int (*ftn)();
  {
  register struct node *stack, *tmp;

  stack = NULL; /* stack initially empty */
  for ( ; ; )
    if (root) {
      tmp = root;       /* move to the left */
      root = root->left;
      tmp->left = stack;        /* stack node */
      stack = tmp;
      }
    else if (stack) {
      root = stack;     /* pop stack */
      stack = stack->left;
      (*ftn)(root);
      root = root->right;
      }
    else
      break;
  }
```

NBS Methane properties package on the IBM-PC under Fortran 3.03. Execution time was 16.5 seconds without the 8087, and 2.5 seconds with the 8087. A time of 90 seconds was obtained for this program on the Xerox 820.

Two results deserve special comment. First, both the original IBM release of Microsoft Fortran and the new 3.03 release (presently under Beta Test) are laughably slow. Note that even the 3.03 version is practically as slow as the 8-bit Fortran of the Xerox 820! This result is presumably due to Microsoft's unwise decision to write their Fortran in Pascal – thus burdening it with all of Pascal's inefficiencies. Even Microsoft's own BASIC compiler is faster! These Fortrans are therefore nearly useless as serious tools for engineering work, and we can only hope that some enterprising firm will fill the need for a fast Fortran for the IBM PC.

Next, the BASIC compiler with the 8087 does not achieve anything like the speed improvement that Fortran does. Whether this is due to inefficiencies in the MicroWare BASIC87 library or to problems with the BASIC compiler itself is an open question.

Sincerely,
Jim Glass
Chris Landis
4747 Orion Avenue, Apt. C
Sherman Oaks, CA 91403

Figure 2.

Machine	Language	Approx. Execution Time Seconds	
CDC-176	FORTRAN	0.8	
IBM-3033	FORTRAN	1.63	
UNIVAC 1100/83	ASCII FORTRAN	12	
TEKTRONIX 4081	FORTRAN	196	
PDP-11/34	FORTRAN	94	
IBM-PC	BASIC Compiler	866	(773)*
Xerox-820	FORTRAN	3386	
IBM-PC	IBM FORTRAN (original)	4440	(1711)*
IBM-PC	FORTRAN 3.03 (new Microsoft release)	3284	(585)**
IBM-PC	BASIC Interpreter	11935	
Apple	BASIC Compiler	5153	
Apple	BASIC Interpreter	16937	

* Using MicroWare 8087 Library
** Using Microsoft 8087 Library

conclusion. The CCS 2810, or at least our elderly example of it, was not kosher by IEEE 696 standards. It was generating sINTA, the interrupt acknowledgement, much too late in the interrupt cycle for the CompuPro board to respond (two clocks later than pSYNC, in fact). Oddly enough, it *was* generating a perfectly good INTA status signal; however, this signal is gated onto the data bus along with the other simulated 8080 status bits, as a compatibility feature. The CompuPro, as you might expect, wouldn't have anything to do with such non-IEEE frippery. It expects sINTA to become valid, along with the other S-100 status bits, when pSYNC comes up. We were able to get the good INTA signal onto the sINTA bus line by cutting one trace and soldering in one jumper on the CCS board.

(That sounds so cool. Just cut a trace, solder a jumper, pooh, no problem. Listen, no diamond cutter ever braced as carefully as we did before cutting that trace. And the sweat was rolling in streams when we powered up afterward. Would our one and only CPU board come up, or would some chip quietly go phhht and leave us without a computer? And it did *not* come up! *PANIC!* Then, looking down from the ceiling, we noticed that we had forgotten to connect the ribbon cable that runs from the CPU board to the CRT. With that hooked up, it ran. Phew.)

CompuPro vs. Itself

We were using one port of our old CompuPro Interfacer II board as a source of test interrupts. A Diablo printer with a keyboard was hooked to it, so we could produce a byte and an interrupt, while a test program reported the results on the CRT. The Interfacer II generated interrupts just as expected, provided we enabled only received-data interrupts. If we enabled transmitted-data interrupts, the test program would report a continuous stream of them, indicating that the Interfacer II was holding its vectored interrupt line in a low state all the time.

This should not have been the case. According to the schematic, the Interfacer II should reset its interrupt signal when it sees sINTA. We pulled the board to see if there was a chip we could change easily. Then the problem became clear: the board didn't match the schematic that came with it. Where there should have been a chip U13, there was only blank green board; the inverted Transmit Buffer Empty signal from the UART went straight to the interrupt line and there *was no reset circuitry* at all! We never did find out whether we got an early board with a late schematic, or vice versa. The Interfacer II isn't in production anymore, and thank goodness we didn't need its interrupts.

CCS vs. The World

The interrupt we really wanted was one from the UART on the 2810 CPU board, to which the CRT is attached. Unfortunately, CCS designed their board assuming that only polled I/O would be used. They used the nice 8250 UART (the IBM PC uses it, too), which has an elaborate structure of prioritized interrupts built into it. However, CCS left the 8250's interrupt pin unconnected. We'd inquired about this when first looking into interrupts a year ago, and were then given instructions on how to access the 8250's interrupt ability. Here, for the information of those who have a 2810 CPU board, is how to do it.

The 8250 generates an interrupt signal on its pin 30. This is a positive-going, CMOS signal. It has to be buffered and inverted before it can be applied to the S-100 bus. Near it on the board is U7, which contains an unused XOR gate (make sure it really is unused); this can be used as an inverter. We were able to connect U5 (the 8250) pin 30 to U7 pin 12, and U7 pin 13 to U7 pin 14 (tying it to +5 volts). Then we ran a rather lengthy jumper from U7 pin 11 up and over the top edge of the board, and all the way down to the solder pad for S-100 line 4, VIO*.

It worked perfectly the first time. During a cold start, our BIOS has to prime

AUGUSTA, Part III
Recursive Descent Compilers

by Edward Mitchell

Edward Mitchell, Box 390145, Mountain View, CA 94039.

This series Copyright © 1983 by Edward Mitchell. The Augusta Compiler is Copyright © 1983 by Computer Linguistics. "Augusta" is a trademark of Computer Linguistics. "Ada" is a registered trademark of the U.S. Department of Defense.

Compilers are computer programs that perform language translation. Typical compilers translate a high-level lanugage, such as BASIC or Pascal into the machine language of the computer that the program will run on (see Figure 1). Part III of this four-part series describes how one type of compiler works.

Compilers as Language Translators

The compiler doesn't have to translate directly into machine code. Some compilers output assembly language that must be assembled later. Others output an intermediate language that, in turn, is compiled to machine code or interpreted at run-time. There are even compilers that translate from one high-level language into another. For example, the RATFOR language is compiled into Fortran.

Augusta programs are compiled into an intermediate language called "pseudo-code" ("p-code" for short). In effect, the p-codes are the machine language of a hypothetical or pseudo-machine. The compiler uses a technique known as "recursive descent parsing" to perform the translation. Compilers, and in particular recursive-descent compilers, are the subject of this article. Included in this part is the first half of the Augusta compiler source listing (see page 20). Part IV contains the rest of the compiler listing and a detailed description of the compiler's operation.

The Compilation Process

Compilation is divided into three steps: (1) lexical analysis, (2) syntactic analysis, and (3) semantic analysis and code generation. Some compilers make several passes through the source program, perhaps performing one of the above steps on each pass. Augusta is a one-pass compiler, so all three steps occur simultaneously.

Lexical analysis carves the source program into logical entities called "tokens." To the computer, a source file is just a long sequence of characters. The lexical analyzer breaks the characters into recognizable groups such as "IF", "THEN", "+", "(", "ALPHA", and so on.

Syntactic analysis or "parsing" ensures that the tokens returned by the lexical analyzer appear in the correct order. The syntax diagrams presented in Part I of this series describes the order of tokens for all valid Augusta programs.

When the syntax analyzer recognizes a valid sequence of tokens, the compiler outputs p-codes. During compilation, these three steps are intermingled. When the syntax analyzer needs a new token, it calls the lexical analyzer. When enough tokens have been read so that the parser can recognize a statement or phrase of the language, then the code is generated.

Lexical Analysis

The lexical analyzer reads the source program character by character. Once it recognizes a valid sequence of characters, it returns the entire group as a token.

The classical approach to designing a lexical analyzer is to draw a *state diagram*. The state diagram is really just a special kind of flowchart where circles replace the usual rectangular boxes and diamonds. The diagram describes what the lexical analyzer should do for each character that it sees. Figure 2 (page 14) shows a diagram to separate digits from letters in a stream of characters like "ABC5347 PWX943ZW1F0". That input would produce the following sequence of tokens: "ABC", "5347", "PWX", "943", "ZW", "1", "F", and "0".

Each circle inside the diagram represents a *state*. The diagram begins with state 1. On seeing the letter "A", the lexical analyzer reads a new character from

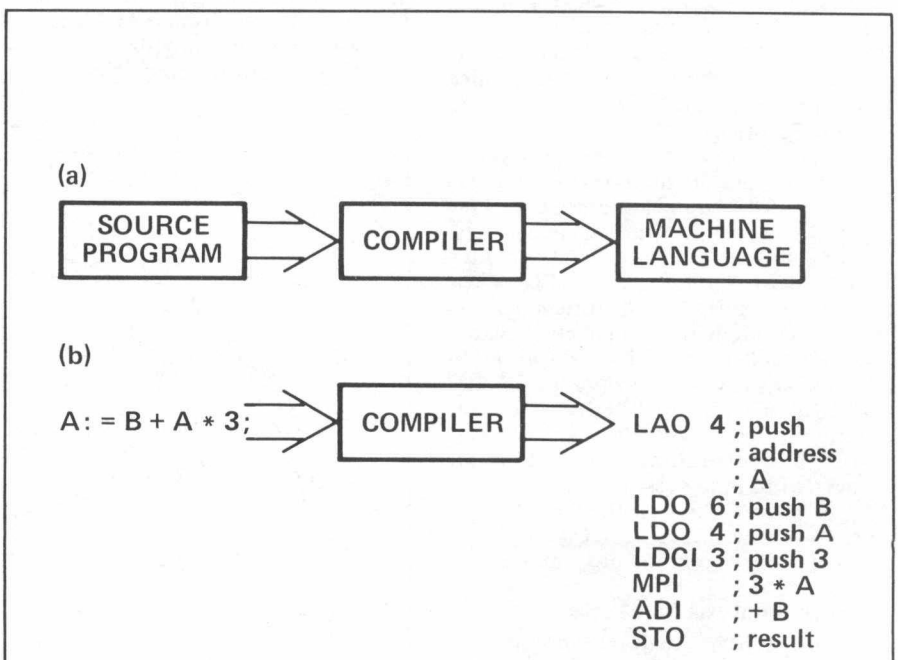


Figure 1.

(a) A compiler translates a high-level language into the machine language of the computer that the program will run on. (b) The translation of a simple Augusta statement into p-codes.

the input and jumps to state 3. The asterisk ("*") indicates that a character is read *before* entering the new state.

The analyzer remains in state 3 as long as it continues to see an alphabetic character. Upon reading "5" from the input, the analyzer jumps to state 4. At this point, it has identified the first complete token "ABC". Returning to state 1, the analyzer is still looking at a "5" so it jumps to state 2 and processes the digit characters. Listing 1 (page 17) shows how the state diagram can be converted to a BASIC program.

A compiler, such as Augusta, must recognize a large number of tokens – many more than just letters and numbers – for example, all of the arithmetic operators, "+", "-", "*", and "/", plus all of the punctuation symbols like "(", ";", ")", and ":=". Some symbols are harder than others to recognize. For example, "/" by itself is the division symbol. But if it is immediately followed by "=", as in "/=", it becomes the "not equal" relational operator. Similarly, some tokens can be quite lengthy. Take the character string, for example. It begins with a double quote (") followed by a large number of characters that are terminated by a trailing quote.

Just as flowcharts are unnecessary for coding many programs, state diagrams are not really needed for many lexical analyzers. Instead, a practiced programmer can directly code an appropriate analyzer fairly quickly. To wit, Augusta's lexical analyzer was written in a single evening and appears in lines 1280 through 1775 of Listing 2 (page 20).

The lexical analysis stage of the compiler will often do additional work to aid the rest of the compilation process. Numbers are converted from their character representation (e.g., "197") into an internal numeric form (e.g., TN=197, where TN is a variable). All identifiers (any token beginning with an alphabetic character) are looked up in the keyword table. If the identifier is a keyword like "AND" "PROCEDURE", or "ELSE", then the lexical analyzer marks the token as a keyword. Unquoted lower case letters are converted to upper case.

In most compilers, the lexical analyzer is a subroutine. At each call, it returns the next token from the program source.

Syntax Analysis

The syntax analyzer or "parser" checks that the tokens appear in the proper order. The "proper order" is determined by the language's syntax which is the set of rules that defines the language. The syntax diagrams in Part I (January 1983) of this series show what valid Augusta statements look like.

How does syntax analysis work? There are actually several techniques for parsing a computer program. Augusta uses a method called recursive-descent parsing, which is a variation of "top-down" parsing. The names "recursive descent" and "top down" come from how the parser builds a *parse tree*. (You don't need to understand the theory in great detail. However, it's convenient for following the basic operation of the parser. I'll keep this theoretical section to a minimum.)

A tree is a special type of data structure that is normally depicted upside down as in Figure 3 (page 15). This implies that the "root" is at the top of the tree and the "branches" are at the bottom. Top-down parsing is so named because it produces a tree-like structure from the top and going downwards as it parses a statement. Compare the syntax diagrams in Figure 4 (page 16) to the parse tree in Figure 3(b) (page 15). The parse tree represents the path that the compiler follows through the syntax diagrams as it parses an expression.

To see how syntax analysis works, we'll construct a parser for the simplified expressions described by the syntax diagrams in Figure 4 (page 16). These diagrams recognize the algebraic hierarchy of operations: "*" and "/" come before "+" and "-".

The syntax diagrams are recursive because one of the definitions is defined in terms of itself. EXPR is defined using SE; SE is defined using FACTOR; and in the case of a parenthesized expression, FACTOR is defined using EXPR.

The diagrams can be translated into a set of procedure calls. A simple implementation of the parser written in Augusta is shown in Listing 3 (page 27). Procedure GETTOKEN (not shown in Listing 3) reads a single token of input and places it in the global variable TOKEN. TOKEN is a character containing "C" if a constant was read, "I" if an identifier was read, or one of the punctuation symbols "(", ")", "*", "/", "+", "-".

To parse the expression, 3*(5+6), GETTOKEN reads the "3". EXPR calls SE to evaluate a "simple expression." SE, in turn, calls FACTOR. Seeing a constant "C", FACTOR reads the next token and returns to SE. Since the token "*" is not one of "+" or "-", control returns to EXPR. But "*" is recognized by EXPR, so EXPR reads the next token, a left parenthesis "(", and calls SE again. SE calls FACTOR where the parenthesis is handled. FACTOR scans past the "(" by calling GETTOKEN and calls EXPR to evaluate the subexpression. After returning from EXPR, FACTOR sees the right parenthesis ")" and returns to SE, which returns to EXPR.

Semantic Analysis

Statements that are syntactically valid may still not make sense. Variable identifiers such as A and B can be multiplied together if and only if A and B are numeric variables. If A and B are strings, then the statement is meaningless. For example,

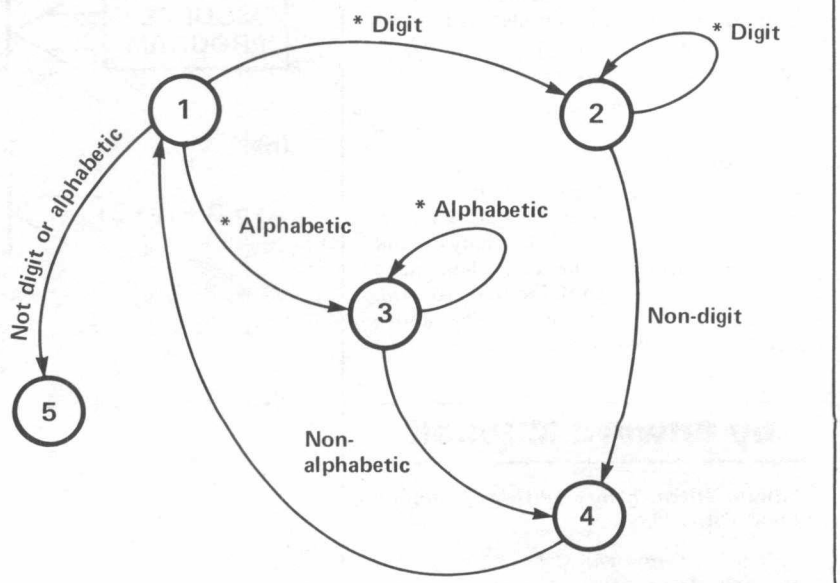


Figure 2.
A simple state diagram to separate letters from digits. See text for details.

(1) A := "String 1";
(2) B := "String 2";
(3) C := A * B;

Statement 3 is syntactically correct but semantically wrong.

Semantic analysis attempts to infer meaning from the statement; e.g., C is to be assigned the quantity of A multiplied by B. Once the compiler understands the statement, it can generate the corresponding p-code translation.

Syntax analysis only cares about finding the correct tokens, such as a constant, an identifier, a keyword, or so on. Semantic analysis, on the other hand, examines the value of the tokens. For example, the identifier "ALPHA" must be looked up in a symbol table to see if it is defined. If it is defined, then the compiler must determine that it can be used correctly in the current context.

Code Generation

Augusta generates code as it does the parsing. An example, using the simple expression parser, shows how this is done.

Let

```
X := 5 * 4;
```

be a simple assignment statement. The compiler, upon seeing an identifier followed by ":=" begins parsing the assignment statement. The symbol "X" is looked up in the symbol table, which tells the compiler that X is an integer variable stored at global offset 10. Augusta generates

```
LDA 10 ; Load address of variable X
```

The compiler next scans across the expression. But note that because Augusta generates code for a stack machine, both operands (the "5" and the "4") must be pushed on the stack before performing the multiplication. This means that the following sequence of p-codes will be created:

```
LDCI 5  ; Load constant 5 on to the
        ; stack
LDCI 4  ; Load constant 4 on to the
        ; stack
MPI     ; Pop top 2 words, multiply
        ; together, and push result
```

Finally, the top of stack value is stored at X, by emitting the instruction

```
STO     ; Store top of stack value in-
        ; to the address contained in
        ; the next word on the stack
```

Listing 4 (page 28) shows the simple expression parser modified to generate p-codes.

Two new procedures, EMITBYTE and EMITWORD, have been created. The parsing routines call EMITBYTE and EMITWORD when they need to output p-codes or p-code parameters. For example, the LDCI opcode is output with the following calls:

```
EMITBYTE (1)  ; – LDCI p-code
              ;   value
EMITWORD (5)  ; – constant 5 fol-
              ;   lows LDCI
```

Code Generated for FOR Loops

The previous section gave an overview of the code generation scheme. We will now look at some specific statements and see what the corresponding p-code translation looks like.

The FOR loop has the form

```
FOR <Identifier> IN
   <Starting Value> . .
   <Ending Value>
LOOP
   <Sequence of Statements>
END LOOP;
```

For example,

```
FOR I IN 1 . . 10
LOOP
   PUTINT(I);
   NEWLINE;
END LOOP;
```

prints the integers 1 throught 10.

For the loop above, the compiler must emit code to set I:=1 at the beginning of the loop to check to determine

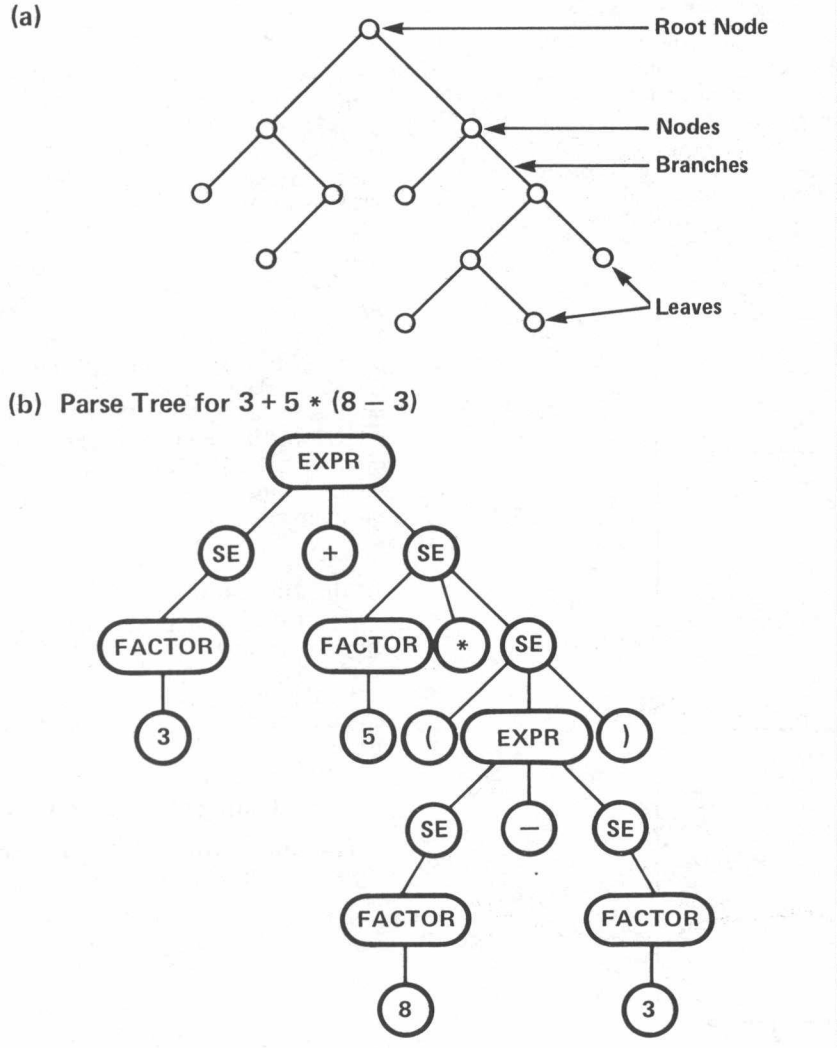


Figure 3.

(a) shows a data structure called a *tree*. Because the tree is depicted upside down, the root is on the top. (b) shows a *parse tree* produced by parsing a simple expression according to the syntax diagrams in Figure 4. Parse trees are the basis of recursive-descent parsing, the technique used by the Augusta compiler.

if I is greater than 10 and to increment I each time the loop is executed. Listing 5 (page 29) shows sample code for the loop shown above.

When the FJP instruction at (1) in Listing 5 is emitted, the compiler does not yet know where the program should jump to. Until the code for the enclosed sequence of statements is generated, the compiler doesn't know how many bytes the FJP instruction should jump past. Instead, it must remember where the FJP instruction is located, generate the remaining code, and then go back and fix the FJP instruction so that it will jump to the correct location. This technique is called "back patching" and is used extensively when compiling long sequences of IF-THEN-ELSEIF and CASE statements.

The WHILE Statement

The WHILE statement is translated into a conditional test, followed by the enclosed statements and ending with a jump back to the conditional test. Listing 6(b) (pages 28 and 29) shows the code produced for WHILE statement shown in 6(a) (page 28).

IF-THEN Statements

Listing 7(a) (page 29) shows sample code for a simple IF-THEN-ELSE statement. Back patching is done for all the forward jump instructions, which includes both the FJP and UJP p-codes in Listing 7(b) (page 29).

"Short circuit" boolean expressions, whether used in assignment statement or as part of an IF-THEN test, use the FJP instruction to jump over portions of the expression code. Listing 8(b) (page 30) shows the code generated for the statements in 8 (a) (page 30).

The CASE Statement

The CASE statement is unusual in that it uses the XJP (indirect jump) opcode to branch to the appropriate section of code. XJP is a seven-byte opcode, having the format

```
XJP w1, w2, w3
```

where w1, w2, and w3 are 16-bit words. XJP uses the value on the top of the stack as an index into a jump table that appears after the code for each of the cases. w1 and w2 are the minimum and the maximum case selections, respectively. For example, in a CASE statement like

```
CASE I IS
   WHEN 7 => ...
   WHEN 11 => ...
   WHEN 19 => ...
   WHEN 23 => ...
END CASE;
```

the minimum selector, w1, is 7, and the maximum selector, w2, is 23. Word w3 is the offset to the indirect jump table. Listing 9 (page 30) shows the basic format for the case statement.

The jump table has the following format,

```
→ UJP          instruction
  address of OTHERS condition
  address of case w1
  address of case w1+1
  address of case w1+2
        :
        .
  address of case w2-1
  address of case w2
```

For the CASE statement shown previously, with selectors 7, 11, 19, and 23, an entry is made in the jump table corresponding to each selector. But the jump table has sixteen entries (maximum selector minus minimum selector or 23 - 7) and so only four entries are accounted for. All other, unused selectors are filled with the address of the UJP instruction at the beginning of the table. Then, when a case value for I is not one of 7, 11, 19, or 23, the program does an automatic jump to the OTHERS clause. The first entry in the table is the address of the code to handle the OTHERS condition. In the event that there is no OTHERS clause, this entry simply jumps to the first instruction after the jump table.

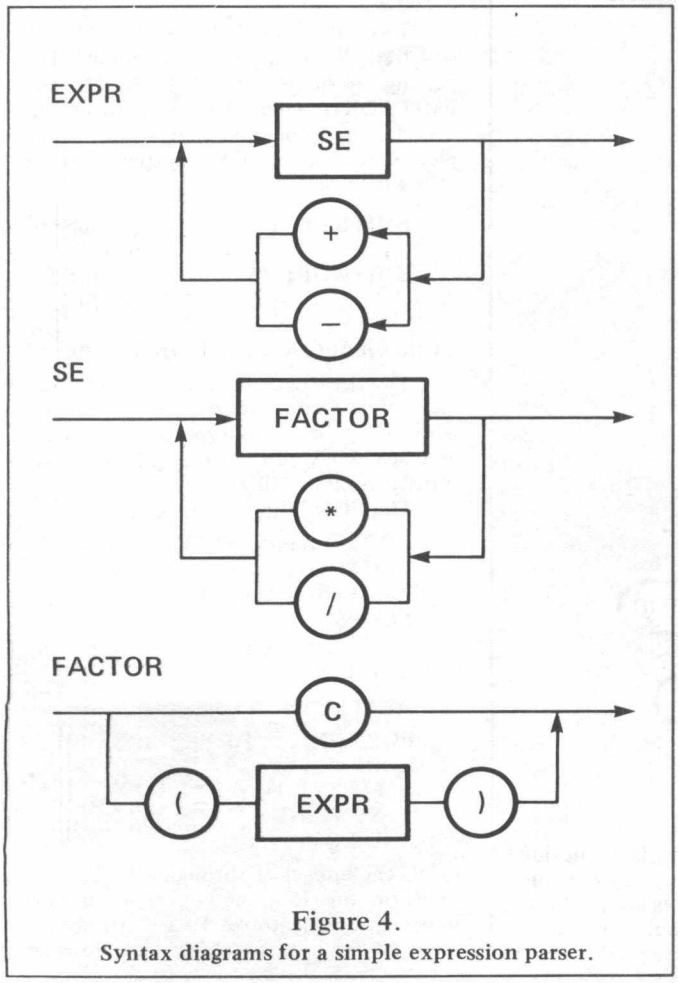


Figure 4.
Syntax diagrams for a simple expression parser.

In the July Issue

Part IV contains the final half of the Augusta compiler source listing. It concludes with some personal comments on the subjects of Ada, the rationale for creating Augusta, the DoD, and Microsoft BASIC. ■■J

(Listing One below)

(Listings 2 through 9 on pages 20 through 30)

Augusta Part III

Listing One

```
10 'SIMPLE LEXICAL ANALYZER
20 '
30 DIGITS$="0123456789"
40 ALPHA$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
50 INPUT "ENTER STRING:  ",S$
60 P=1              ' P POINTS TO NEXT CHAR TO READ
70 GOSUB 800        ' READ CHAR INTO CH$
100 'STATE 1 - DECIDE IF ALPHABETIC OR DIGIT
110 IF INSTR(DIGITS$,CH$) THEN GOSUB 800: GOTO 200
120 IF INSTR(ALPHA$,CH$) THEN GOSUB 800: GOTO 300
130 GOTO 500
200 'STATE 2 - READ IN DIGITS
210 IF INSTR(DIGITS$,CH$) THEN GOSUB 800: GOTO 200
220 GOTO 400
300 'STATE 3 - READ IN ALPHABETIC CHARS
310 IF INSTR(ALPHA$,CH$) THEN GOSUB 800: GOTO 300
320 GOTO 400
400 'STATE 4 - OUTPUT THE EXTRACTED TOKEN
410 PRINT TOKEN$
420 TOKEN$=""
430 GOTO 100
500 'STATE 5 - INVALID CHARACTER STOPS
510 STOP
800 'READ CHARACTER OF INPUT
810 IF P>LEN(S$) THEN CH$=".": RETURN
820 TOKEN$=TOKEN$+CH$     ' ACCUMULATE A TOKEN
830 CH$=MID$(S$,P,1)      ' GET NEXT CHARACTER
840 P=P+1
850 RETURN
```

End Listing One

(Listing Two begins on page 20)

Augusta Part III

Listing Two

Source listing of the Augusta Compiler (Part 1 of 2 parts).

(Text begins on page 13)

Augusta compiler, copyright © 1983 by Computer Linguistics. Permission is granted to individuals to use the compiler for their personal use. Unauthorized distribution is expressly prohibited. Augusta is a trademark of Computer Linguistics.

```
1000 DEFINT A-Z:
      PRINT"Augusta(TM) Compiler V1.1A ":
      PRINT"(C) Copyright 1983 by Computer
      Linguistics":
      PRINT"All rights reserved."
1003 DIM MAP(26), KEY$(33), S$(100),
      TY(20), B$(3), B(3), S(500)
1010 PRINT:
      PRINT"Initializing":
      GOSUB 1780:
      GOSUB 1110:
      SI=1:
      INPUT"Source File ? ",S$:
      GOSUB 1230
1020 INPUT"Code File ? ",C$:
      OPEN"R",#1,C$,128:
      CLOSE 1:
      KILL C$
1040 INPUT"Listing(Y/CR)? ",T1$:
      OPEN"R",#1,C$,128:
      R0=16:
      M0=R0:
      IF T1$="Y" THEN PLST=-1:
      LPRINT LP$;
1050 GOSUB 1290:
      GOSUB 1400:
      PRINT FRE("");" Bytes for Symbols":
      GOSUB 1980
1060 PUT #1, R0:
      BAR=15:
      ID=16
1135 SC=17:
      COMMA=18:
      SEMICOLON=19:
      COLON=20:
      EQGT=21:
      COLONEQ=22:
      DOT=23:
      DOTDOT=24:
      CH=25:
      AT=26
1140 KAND=27:
      KARRAY=28:
      KBEGIN=29:
      KCASE=30:
      KCONST=31:
      KDECLARE=32:
      KELSE=33:
      KELSEIF=34:
      KEND=35:
      KEXIT=36:
      KFOR=37:
      KFUNC=38:
      KIF=39
1145 KIN=40:
      KIS=41:
      KLOOP=42:
      KLAST=43:
      KLEN=44:
      KMOD=45:
      KNOT=46:
      KNULL=47:
      KOF=48:
      KOR=49:
      KOTHERS=50:
      KOUT=51
1150 KPRAGMA=52:
      KPROC=53:
      KRET=54:
      KREVERSE=55:
      KTHEN=56:
      KWHEN=57:
      KWHILE=58
      PNOT=18:
      PADI=19:
      PNGI=20:
      PSBI=21:
      PMPI=22:
      PDVI=23:
      PIND=24:
      PEQUI=25:
      PNEQI=26:
      PLEQI=27:
      PSLDC=61:
      PINCL=80:
      PDECL=81
1210 PLESI=28:
      PGEQI=29:
      PGTRI=30:
      PEQUSTR=31:
      PNEQSTR=32:
      PLEQSTR=33:
      PLESSTR=34:
      PGEQSTR=35:
      PGTRSTR=36:
      PUJP=37:
      PFJP=38:
      PXJP=39

1215 PCLP=40:
      PCGP=41:
      PCSP=42:
      PRET=43:
      PMODI=45:
      PCIP=46:
      PRNP=47:
      PEQP=15:
      PSLDCN1=63:
      PIXA=48

1217 PSLDO=57:
      PSLAO=58:
      PSLLA=59:
      PSLDL0=49:
      PSLDL=60
1220 RETURN
```

```
      FIELD #1, 2 AS T1$, 2 AS T2$, 2 AS T3$,
      2 AS D$,2 AS S$
1070 LSET T1$=MKI$(GC):
      LSET T2$=MKI$(M0):
      LSET T3$=MKI$(PROC):
      LSET D$=MKI$(0):
      LSET S$=MKI$(1113)
1080 PUT #1,1:
      FIELD #1,128 AS D$:
      FOR I=1 TO MB:
      IF B(I)<>0 AND B(I)<>R0 THEN LSET
      D$=B$(I):
      PUT #1, B(I)
1090 NEXT I:
      CLOSE 1:
      PRINT:
      PRINT "Compiled OK":
      PRINT LN;"lines. ";GC-1920;"Bytes":
      GOTO 32767



1110 'INIT
1120 QUOTE$=CHR$(34):
      LEXCH$=ALF$+DIG$+" @$+=-()/:;')(,"+
      QUOTE$+".#!"+CHR$(3)+CHR$(96)+CHR$(9):
      CLST=-1
1130 SQUOTE=0:
      EOL=1:
      C=2:
      LP=3:
      RP=4:
      MUL=5:
      DIV=6:
      ADD=7:
      SUBT=8:
      LES=9:
      LEQ=10:
      GT=11:
      GEQ=12:
      EQ=13:
      NEQ=14:
```

(Continued on page 20, column 2)

```
1160 ADDOPS$=CHR$(ADD)+CHR$(SUBT):
      MULOPS$=CHR$(MUL)+CHR$(DIV)+CHR$(KMOD):
      LOGICALOPS$=CHR$(KAND)+CHR$(KOR)
1165 UNARYOPS$=CHR$(ADD)+CHR$(SUBT)+CHR$(KNOT):
      RELOPS$=CHR$(LES)+CHR$(LEQ)+CHR$(GT)+
      CHR$(GEQ)+CHR$(EQ)+CHR$(NEQ)
1170 DECLPARTS$=CHR$(ID)+CHR$(KPROC)+
      CHR$(KFUNC)+CHR$(KPRAGMA)
1180 STMTS$=CHR$(KWHILE)+CHR$(KFOR)+
      CHR$(KLOOP)+CHR$(KDECLARE)+CHR$(KBEGIN)+
      CHR$(KEXIT)+CHR$(KRET)+CHR$(KIF)
1185 STMTS$=STMTS$+CHR$(KCASE)+
      CHR$(KNULL)+CHR$(ID)+CHR$(KPRAGMA)
1190 LN=1:
      EOI=0:
      LL=0:
      CPROC=0:
      PROC=0:
      GC=1920:
      VLOC=VARPTR(V):
      VLOC1=VLOC+1:
      TSTR=0:
      TINT=1:
      TCHR=2:
      TBOL=4:
      FMSZ=14
1200 PLDCI=1:
      PLDL=2:
      PLLA=3:
      PLDB=4:
      PLDO=5:
      PLAO=6:
      PDUP=7:
      PLOD=8:
      PLDA=9:
      PPOP=10:
      PSTO=11:
      PSINDO=12:
      PLCA=13:
      PSAS=14:
      PAND=16
1205 POR=17:
```

(Continued on page 20, column 3)

```
1230 'OPEN SRC
1240 SI=SI+1:
      OPEN"I",#SI,S$:
      RETURN



1250 'LC TO UC
1260 IF INSTR(LC$,CH$) THEN CH$=CHR$(ASC(CH$)-32)
1270 RETURN



1280 'GETLINE
1290 LN=LN+1:
      IF EOF(SI) THEN CLOSE SI:
      SI=SI-1:
      IF SI>1 AND PLST THEN LPRINT
      TAB(26);"END OF INCLUDE"
1300 IF SI=1 THEN EOI=-1:
      RETURN
1310 LINE INPUT #SI, BUF$
1320 IF PLST=0 GOTO 1330 ELSE LPRINT
      USING "##### #### ###### ###### ";LN,
      CPROC,CP,OFST;:
      LPRINT BUF$
1325 IF (LN MOD 60)=0 THEN LPRINT CHR$(12);:
      LPRINT:
      LPRINT
1330 IF CLST<>0 THEN PRINT BUF$ ELSE IF
      (LN AND 63)=63 THEN PRINT LN;"..."
1340 IF LEN(BUF$)=0 GOTO 1290 ELSE
      BUF$=BUF$+CHR$(3):
      B=1:
      WHILE MID$(BUF$,B,1)=" ":
      B=B+1:
      WEND:
      CH$=MID$(BUF$,B,1):
      B=B+1:
      RETURN
```

(Continued on next page)

Augusta Part III

Listing Two

(Continued, text begins on page 13)

```
1360 'GETCH
1370 LSET CH$=MID$(BUF$,B,1):
       B=B+1:
       RETURN
1380 B=B+1
1390 RETURN



1400 'GETSYM
1410 OLDB=B:
       GOSUB 1250:
       I=INSTR(LEXCH$,CH$):
       IF I=0 THEN E=1:
       GOTO 5020
1420 IF I<27 THEN GOSUB 1460:
       GOTO 1430
1423 IF I<42 THEN ON I-26 GOSUB 1500,
       1500,1500,1500,1500,1500,1500,1500,1500,
       1500,1700,1770,1720,1520,1600:
       GOTO 1430
1427 ON I-41 GOSUB 1530,1620,1640,1660,
       1680,1540,1750,1570,1560,1550,1730,1580,
       1695,1710,1450,1440,1775
1430 IF EOI THEN E=12:
       GOTO 5020 ELSE IF OLDB=B GOTO 1410 ELSE
       LSET TT$=CHR$(T):
       RETURN
1440 T=SQUOTE:
       GOSUB 1360:
       RETURN
1450 GOSUB 1290:
       OLDB=B:
       RETURN
1460 S$="":
       WHILE INSTR(AN$,CH$):
       IF CH$<>"_" THEN S$=S$+CH$
1470 GOSUB 1370:
       GOSUB 1260:
       IF CH$="." THEN T=DOTDOT:
       GOSUB 1360
1590 RETURN
1600 GOSUB 1360:
       IF CH$=")" THEN T=EQGT:
       GOSUB 1360 ELSE T=EQ
1610 RETURN
1620 GOSUB 1360:
       IF CH$="=" THEN T=LEQ:
        GOSUB 1360 ELSE T=LES
1630 RETURN
1640 GOSUB 1360:
       IF CH$="=" THEN T=GEQ:
        GOSUB 1360 ELSE T=GT
1650 RETURN
1660 GOSUB 1360:
       IF CH$="=" THEN T=NEQ:
        GOSUB 1360 ELSE T=DIV
1670 RETURN
1680 GOSUB 1360:
       IF CH$="=" THEN T=COLONEQ:
       GOSUB 1360 ELSE T=COLON
1690 RETURN
1695 T=BAR:
       GOSUB 1360:
       RETURN
1700 WHILE CH$=" ":
       LSET CH$=MID$(BUF$,B,1):
       B=B+1:
       WEND:
       OLDB=B:
       RETURN
1710 T=BAR:
       GOSUB 1360:
       RETURN
1720 T=MUL:
       GOSUB 1360:
       RETURN
1730 I1=INSTR(B,BUF$,QUOTE$):
       IF I1=0 THEN E=10:
       GOTO 5020
1740 S$=MID$(BUF$,B,I1-B):
       T=SC:
       B=I1+1:
1795 D$=S$:
       LOC1=0:
       LOC2=0:
       V=0:
       VLOC1=0:
       VLOC=0:
       TN=0:
       TT$=S$:
       HASH=0:
       ID$=S$:
       BUF$=S$:
       T1=0:
       T2=0


1800 NKEY=33:
       SSP=1:
       MB=3:
       FOR I=0 TO MB:
       B$(I)=SPACE$(128):
       B(I)=0:
       NEXT I


1820 OPEN "I",#1,"KEYWORDS.TXT":
       LINE INPUT #1,LC$:
       T1=1:
       WHILE T1>0:
       INPUT #1,T1:
       LP$=LP$+CHR$(T1):
       WEND
1830 INPUT#1,DIG$,HDIG$,ALF$,LC$,AN$
1840 FOR I=1 TO 26:
       INPUT #1,MAP(I):
       NEXT I


1850 I=1:
       INPUT #1,ID$,TYPE,KIND,PINFO,CONST,
       OBJSZ,ADDR,LL:
       IF ID$<>"$END$" THEN ID$=ID$+SPACE$(8-
       LEN(ID$)):
       GOSUB 3850:
       GOTO 1850
```

```
      WEND:
      IF LEN(S$)>8 THEN S$=LEFT$(S$,8)
1480 ID$=S$+SPACE$(8-LEN(S$)):
      GOSUB 1890:
      RETURN



1490 'DIGITS
1500 TN=0:
      I1=10
1510 WHILE INSTR(HDIG$,CH$):
      TN=TN*I1+INSTR(HDIG$,CH$)-1:
      GOSUB 1360:
      WEND
1515 IF CH$="#" THEN I1=TN:
      TN=0:
      GOSUB 1360:
      GOTO 1510 ELSE T=C:
      RETURN
1520 T=ADD:
      GOSUB 1360:
      RETURN
1530 T=SUBT:
      GOSUB 1360:
      IF CH$="-" THEN GOSUB 1280:
      OLDB=B:
      RETURN ELSE RETURN
1540 T=SEMICOLON:
      GOSUB 1360:
      RETURN
1550 T=COMMA:
      GOSUB 1360:
      RETURN
1560 T=LP:
      GOSUB 1360:
      RETURN
1570 T=RP:
      GOSUB 1360:
      RETURN
1580 T=DOT:
      GOSUB 1360:
```

(Continued on page 22, column 2)

```
      GOSUB 1360:
      RETURN
1750 GOSUB 1360:
      GOSUB 1360:
      IF CH$<>"'" THEN E=11:
      GOTO 5020
1760 GOSUB 1360:
      GOSUB 1930:
      TN=ASC(MID$(S$,2,1)):
      T=CH:
      RETURN
1770 T=AT:
      GOSUB 1360:
      RETURN
1775 GOSUB 1360:
      OLDB=B:
      RETURN



1780 'READ DATA
1790 CH$=" ":
      B=0:
      LB=0:
      AN$=CH$:
      LC$=CH$:
      S$=CH$:
      T=0:
      T0=0:
      X=0:
      SP=0:
      TSP=0:
      LEXCH$=S$:
      CP=0:
      CB=0:
      W=0:
      I=0:
      R2=0:
      R1=0:
      T3=0:
      R0=16
```

(Continued on page 22, column 3)

```
1860 IF EOF(1) GOTO 1880 ELSE INPUT #1,T$:
      IF LEN(T$)>8 THEN T$=LEFT$(T$,8)
1870 T$=T$+SPACE$(8-LEN(T$)):
      KEY$(I)=T$:
      I=I+1:
      GOTO 1860



1880 CLOSE 1:
      KEY$(0)=" ":
      KEY$(NKEY)=" ":
      RETURN



1890 'LOOKUP KEYWORD
1900 HASH=MAP(INSTR(ALF$,LEFT$(ID$,1)))
1910 IF KEY$(HASH)=ID$ THEN T=HASH+26
      ELSE IF ASC(KEY$(HASH))<>ASC(ID$) THEN
      T=ID ELSE HASH=HASH+1:
      GOTO 1910
1920 RETURN



1930 'GET S$
1940 S$=MID$(BUF$,OLDB-1,B-OLDB):
      RETURN
1950 IF T0=T THEN RETURN
1955 E=4:
      GOSUB 5110:
      PRINT "REENTER+ ";:
      LINE INPUT T$:
      BUF$=LEFT$(BUF$,B-1)+T$+CHR$(3):
      GOSUB 1360:
      GOSUB 1400:
      GOTO 1950
1960 IF T0=T THEN GOSUB 1400:
      RETURN ELSE GOSUB 1950:
      GOSUB 1400:
      RETURN
```

(Continued on next page)

Augusta Part III

Listing Two

(Continued, text begins on page 13)

```
1970 'COMPILATION
1980 GOSUB 2770
1990 IF T=KPROC THEN GOSUB 1400:
       GOSUB 2010:
       TO=SEMICOLON:
       GOSUB 1950
2000 RETURN



2010 'PARSE PROC
2020 GOSUB 5200
2030 KIND=2:
       PROC=PROC+1:
       CPROC=PROC:
       ADDR=PROC:
       X=ADDR:
       GOSUB 4280:
       GOSUB 3850:
       GOSUB 1400
2040 OFST=-FMSZ:
       IF T=KIS GOTO 2060
2050 GOSUB 2100:
       TO=KIS:
       GOSUB 1950



2060 'IS
2070 X=-(OFST+FMSZ):
       GOSUB 4280:
       GOSUB 1400:
       OFST=0:
       MXOF=0:
       GOSUB 2440:
       W=PRET:
       GOSUB 3990:
       GOSUB 5300:
       RETURN
2235 T1$=MID$(T1$,9):
       OFST=OFST-2:
       WEND
2240 RETURN



2250 'SUBTYPEINDICATIONUNIT
2260 GOSUB 3890:
       IF KIND<>4 THEN E=8:
       GOTO 5020 ELSE IF PINFO=0 THEN KIND=1
       ELSE KIND=5
2280 IF TYPE<>0 THEN GOSUB 1400:
       RETURN



2285 GOSUB 2300:
       IF OBJSZ>255 THEN E=15:
       GOTO 5020 ELSE RETURN



2290 'GET C
2293 IF T<>ID THEN GOTO 2297 ELSE T8=TYPE:
       T3=KIND:
       T4=PINFO:
       T5=CONST:
       T6=OBJSZ:
       T7=LL



2294 GOSUB 3890:
       IF KIND=0 AND TYPE=1 THEN T=C:
       T2=CONST
2295 TYPE=T8:
       KIND=T3:
       PINFO=T4:
       CONST=T5:
       OBJSZ=T6:
       LL=T7
2297 TO=C:
       GOSUB 1960:
       RETURN
2400 TO=KIS:
       GOSUB 1960
2410 X=-(OFST+FMSZ):
       GOSUB 4280:
       OFST=0:
       MXOF=0:
       GOSUB 2440:
       GOSUB 5300:
       RETURN



2440 'BODYPART
2450 IF INSTR(DECLPARTS$,TT$) THEN GOSUB 2480
2460 CB=GC:
       CP=0:
       GOSUB 2790
2470 RETURN



2480 'DECLPART
2490 IF T=ID THEN T1$=ID$:
       K1=5:
       GOSUB 2560:
        GOTO 2540
2500 IF T=KPROC THEN GOSUB 1400:
        GOSUB 2010:
        GOTO 2540
2510 IF T=KFUNC THEN GOSUB 1400:
        GOSUB 2340:
        GOTO 2540
2520 IF T=KPRAGMA THEN GOSUB 2770:
       GOTO 2550



2530 E=3:
       GOTO 5020
2540 GOSUB 3420
2550 IF INSTR(DECLPARTS$,TT$) GOTO 2480
       ELSE GOSUB 4990:
       RETURN
```

```
2100 'PROCFORMALPART
2110 T2$="":
      T0=LP:
      GOSUB 1960
2120 GOSUB 2160:
      IF T=SEMICOLON THEN GOSUB 1400:
      GOTO 2120
2130 T0=RP:
      GOSUB 1960:
      FOR I=OFST TO -FMSZ-2 STEP 2:
      T1$=LEFT$(T2$,17):
      T2$=MID$(T2$,18):
      IF (LEN(S$(SSP))+17)>255 THEN SSP=SSP+1
2140 S$(SSP)=LEFT$(T1$,14)+MKI$(I)+
      RIGHT$(T1$,1)+S$(SSP):
      NEXT I
2150 RETURN



2160 'PROCPARAMDECL
2170 T1$=""
2180 T0=ID:
       GOSUB 1950:
      T1$=T1$+ID$:
      GOSUB 1400
2190 IF T=COMMA THEN GOSUB 1400:
       GOTO 2180
2200 T0=COLON:
      GOSUB 1960:
      P1=1:
      IF T=KOUT THEN P1=2:
      GOSUB 1400:
      GOTO 2220
2210 IF T=KIN THEN GOSUB 1400
2220 GOSUB 2250:
      PINFO=P1
2230 WHILE LEN(T1$)>0:
      T2$=T2$+LEFT$(T1$,8)+CHR$(TYPE)+
      CHR$(KIND)+CHR$(PINFO)+MKI$(CONST)+
      CHR$(OBJSZ)+MKI$(0)+CHR$(LL)
```

(Continued on page 24, column 2)

```
2300 'OBJSZ
2310 GOSUB 1400:
      IF T<>LP GOTO 2330 ELSE GOSUB 1400
2320 GOSUB 2290:
      OBJSZ=TN+1:
      T0=RP:
      GOSUB 1960
2330 RETURN



2340 'PARSEFUNC
2350 GOSUB 5200:
      KIND=3:
      PROC=PROC+1:
      CPROC=PROC:
      ADDR=PROC:
      X=ADDR:
      GOSUB 4280:
      GOSUB 3850:
      X=SSP:
      GOSUB 4280:
      X=LEN(S$(SSP))
2355 GOSUB 4280:
      GOSUB 1400



2370 OFST=-FMSZ:
      IF T=LP THEN GOSUB 2100
2380 T0=KRET:
      GOSUB 1960:
      GOSUB 2250:
      GOSUB 4300:
      T2=X:
      GOSUB 4300:
      T1=X:
      T3=LEN(S$(T1)):
      IF KIND<>5 OR OBJSZ<>2 THEN E=16:
      GOTO 5020
2385 S$(T1)=LEFT$(S$(T1),T3-T2+8)+
      CHR$(TYPE)+MID$(S$(T1),T3-T2+10)
```

(Continued on page 24, column 3)

```
2560 'OBJDECL
2570 GOSUB 1400
2580 IF T=COMMA THEN GOSUB 1400:
      T0=ID:
      GOSUB 1950:
      T1$=T1$+ID$:
       GOTO 2570
2590 T0=COLON:
      GOSUB 1960



2600 IF T=KCONST GOTO 2650
2610 IF T=KARRAY GOTO 2700
2620 GOSUB 2250:
      OBJSIZE=OBJSZ
2630 PINFO=0:
      KIND=K1:
      WHILE LEN(T1$)>0:
      ID$=LEFT$(T1$,8):
      T1$=MID$(T1$,9):
      ADDR=OFST:
      OFST=OFST+OBJSIZE:
      GOSUB 3850:
      WEND
2640 RETURN



2650 'CONSTANT
2670 K1=0:
      OBJSIZE=0:
      GOSUB 1400:
      T0=COLONEQ:
      GOSUB 1960:
      IF T=ID THEN GOSUB 3890:
      GOTO 2690 ELSE IF T=SUBT THEN T1=-1:
      GOSUB 1400 ELSE T1=1
2680 CONST=TN*T1:
      IF T=C THEN TYPE=1 ELSE TYPE=2
2690 GOSUB 1400:
      GOTO 2630
```

(Continued on next page)

Augusta Part III

Listing Two

(Continued, text begins on page 13)

```
2700 'ARRAY
2710 K1=1:
       GOSUB 1400:
       T0=LP:
       GOSUB 1960:
       T2=TN:
       GOSUB 2290:
       T0=RP:
       GOSUB 1960:
       T0=KOF:
       GOSUB 1960
2750 GOSUB 2250:
       CONST=T2:
       OBJSIZE=(T2+1)*OBJSZ:
       IF T2<0 OR T2>16383 THEN E=15:
       GOTO 5020 ELSE GOTO 2630


2770 'PRAGMA
2780 IF T<>KPRAGMA THEN RETURN ELSE GOSUB 4830:
       GOSUB 1280:
       GOSUB 1400:
       GOTO 2780


2790 'STMT
2800 T0=KBEGIN:
       GOSUB 1960:
       GOSUB 2810:
       T0=KEND:
       GOSUB 1960:
       RETURN


2810 'SEQOFSTMTS
2820 I=INSTR(STMTS$,TT$)
2825 IF I=0 THEN RETURN ELSE ON I GOSUB
       4320,4320,4320,2850,2850,2890,2930,2970,
       4630,2830,3440,2770:
       GOTO 2820
2930 'RETURN
2940 GOSUB 1400
2950 IF T<>SEMICOLON THEN GOSUB 3100:
       TSP=TSP-1:
       W=PRNP ELSE W=PRET
2960 GOSUB 3990:
       GOSUB 3420:
       RETURN



2970 'IF
2980 LUJP=0
2990 GOSUB 1400:
       GOSUB 3100:
       GOSUB 4930:
       W=PFJP:
       GOSUB 3990:
       X=CP:
       GOSUB 4280:
       GOSUB 4030:
       X=LUJP:
       GOSUB 4280
2995 T0=KTHEN:
       GOSUB 1960:
       GOSUB 2810:
       GOSUB 4300:
       LUJP=X



3000 IF T=KEND THEN GOSUB 3040:
       GOTO 3030
3010 IF T=KELSEIF THEN GOSUB 3060:
       GOSUB 3040:
       GOTO 2990
3020 T0=KELSE:
       GOSUB 1960:
       GOSUB 3060:
       GOSUB 3040:
       X=LUJP:
       GOSUB 4280:
       GOSUB 2810:
       GOSUB 4300:
       LUJP=X
3100 'EXPR
3110 GOSUB 3190:
       LFJP=0:
       PREV=0
3120 IF INSTR(LOGICALOPS$,TT$)=0 THEN
       IF PREV GOTO 3180 ELSE RETURN
3125 X=T:
       GOSUB 1400:
       IF (X=KAND AND T=KTHEN) THEN X=KAND+
       KTHEN ELSE IF (X=KOR AND T=KELSE) THEN
       X=KOR+KELSE
3130 IF PREV<>0 THEN IF PREV<>X THEN E=10:
       GOTO 5020
3140 IF X<>KAND AND X<>KOR GOTO 3160
3145 GOSUB 4280:
       GOSUB 3190:
       IF (TY(TSP)<>TBOL) OR (TY(TSP)<>TY(TSP-
       1)) THEN E=9:
       GOTO 5020
3147 TSP=TSP-1:
       GOSUB 4300:
       PREV=X:
       IF X=KAND THEN W=PAND ELSE W=POR
3150 GOSUB 3990:
       GOTO 3120
3160 GOSUB 4280:
       T1=X:
       W=PDUP:
       GOSUB 3990:
       IF T1=KAND+KTHEN THEN W=PFJP ELSE W=PNOT:
       GOSUB 3990:
       W=PFJP
3170 GOSUB 3990:
       W=LFJP:
       LFJP=CP:
       GOSUB 4030:
       GOSUB 1400:
       X=LFJP:
       GOSUB 4280:
       GOSUB 3190
```

(The Augusta Compiler listing will be continued in the July 1983 issue.)

```
2830 'NULL
2840 GOSUB 1400:
       GOSUB 3420:
       RETURN


2850 'BLOCK
2860 X=OFST:
       GOSUB 4280:
       OFST=OFST+2:
       GOSUB 5400:
       IF T=KDECLARE THEN GOSUB 1400:
       GOSUB 2480
2880 GOSUB 2790:
       GOSUB 5500:
       GOSUB 5700:
       GOSUB 4300:
       OFST=X:
       GOSUB 3420:
       RETURN


2890 'EXIT
2900 IF LPFLG=0 THEN E=14:
       GOTO 5020
2910 GOSUB 1400:
       IF T=SEMICOLON THEN W=PUJP:
       GOSUB 3990:
       GOTO 2925
2920 T0=KWHEN:
       GOSUB 1960:
       GOSUB 3100:
       GOSUB 4930:
       W=PNOT:
       GOSUB 3990:
       W=PFJP:
       GOSUB 3990
2925 W=XITJP:
       XITJP=CP:
       GOSUB 4030:
       GOSUB 3420:
       RETURN
```

(Continued on page 26, column 2)

```
3030 T0=KEND:
       GOSUB 1960:
       T0=KIF:
       GOSUB 1960:
       GOSUB 3080:
       GOSUB 3420:
       RETURN


3040 'FIX FJP
3050 GOSUB 4300:
       T1=CP:
       CP=X:
       W=T1-X-2:
       GOSUB 4030:
       CP=T1:
       RETURN


3060 'GEN UJP
3070 W=PUJP:
       GOSUB 3990:
       W=LUJP:
       LUJP=CP:
       GOSUB 4030:
       RETURN


3080 'FIXUP
3090 T2=CP:
       WHILE LUJP<>0:
       CP=LUJP:
       GOSUB 4010:
       LUJP=W:
       W=T2-CP-2:
       GOSUB 4030:
       WEND:
       CP=T2:
       RETURN
```

(Continued on page 26, column 3)

Augusta Part III

Listing Three

Simple expression parser.

```
PROCEDURE PARSEEXPR IS

PROCEDURE EXPR IS

  PROCEDURE SE IS

    PROCEDURE FACTOR IS
    BEGIN
      IF TOKEN='C' THEN
        GETTOKEN;          -- SCAN TO NEXT TOKEN
        RETURN;            -- RETURN TO CALLER
      END IF;
      IF TOKEN/='(' THEN ERROR ("SYNTAX ERROR");
      ELSE
        GETTOKEN;
        EXPR;
        IF TOKEN/=')' THEN ERROR (") EXPECTED");
        END IF;
      END IF;
    END; -- OF FACTOR

  BEGIN
    LOOP
      FACTOR;
      IF TOKEN='+' THEN
          NULL;            -- Do nothing for now
      ELSEIF TOKEN='-' THEN
          NULL;
      ELSE
          RETURN;
      END IF;
      GETTOKEN;
    END LOOP;
  END; -- OF SE

BEGIN
  LOOP
    SE;
    IF TOKEN='*' THEN
        NULL;
    ELSEIF TOKEN='/' THEN
        NULL;
    ELSE
        RETURN;
    END IF;
    GETTOKEN;
  END LOOP;
END;  -- OF EXPR

BEGIN
    GETTOKEN;
    EXPR;
END;
```

End Listing Three

(Listing Four begins on page 28)

Augusta Part III

Listing Four

Simple expression parser with code generation.

(Text begins on page 13)

```
PROCEDURE PARSEEXPR IS

PROCEDURE EXPR IS

  PROCEDURE SE IS

    PROCEDURE FACTOR IS
    BEGIN
      IF TOKEN='C' THEN
        EMITBYTE( 1 );    -- LDCI
        EMITWORD( TN );   -- the actual constant
        GETTOKEN;         -- SCAN TO NEXT TOKEN
        RETURN;           -- RETURN TO CALLER
      END IF;
      IF TOKEN/='(' THEN ERROR ("SYNTAX ERROR");
      ELSE
        GETTOKEN;
        EXPR;
        IF TOKEN/=')' THEN ERROR (") EXPECTED");
        END IF;
      END IF;
    END; -- OF FACTOR



  BEGIN
    LOOP
      FACTOR;
      IF TOKEN='+' THEN
          EMITBYTE( + );       -- ADI
      ELSEIF TOKEN='-' THEN
          EMITBYTE( - );       -- SBI
      ELSE
          RETURN;
      END IF;
      GETTOKEN;
    END LOOP;
  END; -- OF SE



BEGIN
  LOOP
    SE;
    IF TOKEN='*' THEN
        EMITBYTE( * );         -- MPI
    ELSEIF TOKEN='/' THEN
        EMITBYTE( / );         -- DIVI
    ELSE
        RETURN;
```

```
(b)
     LLA        4          ; Load Address of I
     SLDC1                 ; Short load a constant 1
     STO                   ; I := 1;
¦--> LDL        4          ; put I on stack
¦    SLDC10                ; Short load 10 on stack
¦    LEQI                  ; compare I <= 10
¦ (1)FJP        ---        ; Jump if I > 10
¦ ¦<-----------<-
¦ ¦    .
¦ ¦    .
¦ ¦    .
¦ ¦ <code for enclosed sequence of statements>
¦ ¦    .
¦ ¦    .
¦ ¦    .
¦ ¦  INCL 4               ; Set I := I + 1
¦ ¦  UJP        ---       ; Jump back to comparison
¦<-------------
  ¦
  ¦--> <first instruction after the loop>
```

End Listing Five

Augusta Part III

Listing Six

(b) shows the code generated for the WHILE statement shown in (a).

```
(a)
     PROCEDURE DEMO IS
       I : INTEGER;

     BEGIN
       I:=1;
       WHILE I<10
       LOOP
         I:=I+1;
       END LOOP;
     END;

(b)

     SLA0       0          ; Load address of I
     SLDC1                 ; Short load constant 1
     STO                   ; Store, I:=1
  ¦->SLD0       0          ; Load value of I
  ¦  SLDC10                ; and compare to 10
  ¦  LESI                  ; Push -1 if >10
```

```
    END IF;
    GETTOKEN;
  END LOOP;
END;  -- OF EXPR



PROCEDURE STMT IS
BEGIN
  IF TOKEN/='I' THEN ERROR ("ID EXPECTED");
  ELSEIF TOKEN/='=' THEN ERROR("= EXPECTED");
  ELSE
    GETTOKEN;
    EXPR;
    IF TOKEN/=';' THEN ERROR ("; EXPECTED");
  END IF;
END;

BEGIN
  GETTOKEN;
  STMT;
END;
```

End Listing Four

Augusta Part III

Listing Five

Code generated for the FOR loop. At (a) is a sample FOR loop, and at (b) is the corresponding p-code translation.

```
(a)  FOR I IN 1..10
     LOOP
       .
       .
       .
       <sequence of statements>
       .
       .
       .
     END LOOP;
```

(Continued on page 28, column 2)

```
!<-- FJP      ----        ; Jump if I>10
! !  SLAO      0          ; Load address of I
! !  SLDO      0          ; Load value of I
! !  SLDC1                ; and add 1 to it
! !  ADI
! !  STO
! !<-UJP      ---         ; Jump back for more
!
!--->
```

End Listing Six

Augusta Part III

Listing Seven

Code generation for an IF-THEN-ELSE. (a) shows the IF-THEN-ELSE statement and (b) shows the corresponding compiled code.

```
(a)
     IF A=B THEN
        C=A*2;
     ELSE
        C=B+A;
     END IF;

(b)
     LDL        0          ; Load value of A
     LDL        2          ; Load value of B
     EQUI                  ; Pop top 2 words and compare, push TRUE/FALSE
!<-- FJP        --         ; Jump if they are not equal to do ELSE part
!    LLA        4          ; Load address of C
!    LDL        0          ; Load value of A
!    SLDC2                 ; Load constant 2 to multiply by
!    MPI                   ; Multiply A*2
!    STO                   ; and set C:=A*2
! !- UJP        --         ; jump around the ELSE part
! !                        ; Following code is ELSE part
!--> LLA        4          ; Load address of C
  !  LDL        2          ; Load value of B
  !  LDL        0          ; And load value of A
  !  ADI                   ; Add them together
  !  STO                   ; and store result
  !-->
```

End Listing Seven

(Listing Eight begins on page 30)

Augusta Part III

Listing Eight

"Short-circuit" conditions are provided to evaluate only a portion of an expression. For example, the second expression (J/N > 50), will only be evaluated if the first part tests as true. Augusta provides two short circuit operators – AND THEN and OR ELSE.

(a)

```
IF (N /= 0) AND THEN ( J / N > 50) THEN I:=0;
```

(b)

```
     SLDO      8          ; Get value of N
     SLDC0                ; load constant 0
     NEQI                 ; see if not equal
     DUP                  ; duplicate top of stack value
  ¦<-FJP       ----       ; Jump around if equal
  ¦  SLDO      10         ; Load J
  ¦  SLDO      8          ; Load N
  ¦  DVI                  ; J / N
  ¦  SLDC      50         ; Load 50
  ¦  GTRI                 ; > Comparison
  ¦  AND                  ; AND with previous expression
¦-¦->FJP       ----       ; and jump if false
¦    SLA0      0          ; Load address of I
¦    SLDC0                ; Load constant 0
¦    STO                  ; and store result
¦
¦-->
```

End Listing Eight

Augusta Part III

Listing Nine

(b) shows the code generated for the CASE statement shown in (a). The text describes the XJP instruction in greater detail.

(a)

```
    CASE I OF
      WHEN 7  =>  <statements 1>
      WHEN 11 =>  <statements 2>
      WHEN 19 =>  <statements 3>
      WHEN 23 =>  <statements 4>
    END CASE;
```

(b)

```
     LDL       2          ; Load value of I
     XJP    ___, ___, ___ ; Jump indirectly, using top of stack
             w1   w2   w3  ; value,-w1, as an index in to the
                    ¦      ; jump table at offset w3
                    ¦      ;
  ¦<-----------------
  ¦
  ¦ ¦----> <code for statements 1>
  ¦ ¦                      ; This is case 7
  ¦ ¦      UJP ___         ; Jump past the jump table
  ¦ ¦
  ¦ ¦ ¦--> <code for statements 2>
  ¦ ¦ ¦                    ; This is case 11
  ¦ ¦ ¦    UJP ___         ; Jump past the jump table
  ¦ ¦ ¦
  ¦ ¦ ¦    <code for statements 3>
  ¦ ¦ ¦                    ; This is case 19
  ¦ ¦ ¦    UJP ___         ; Jump past the jump table
  ¦ ¦ ¦
  ¦ ¦ ¦    <code for statements 4>
  ¦ ¦ ¦                    ; This is case 23
  ¦ ¦ ¦    UJP ___         ; Jump past the jump table
  ¦ ¦ ¦
  ¦ ¦ ¦
  ¦ ¦ ¦                    ; The jump table follows
  ¦------> UJP <-----¦
    ¦ ¦    ___ -->¦  ¦     ; address of OTHERS condition
    ¦ ¦           ¦  ¦
    ¦<---- ___    ¦  ¦     ; address of case 7
      ¦           ¦  ¦
      ¦    ___ ---->¦     ; address of case 8 (pointer to OTHERS)
      ¦           ¦  ¦
      ¦    ___ ---->¦     ; address of case 9 (pointer to OTHERS)
      ¦           ¦
      ¦    ___ ---->¦     ; address of case 10 (to OTHERS)
      ¦           ¦
      ¦<-- ___    ¦       ; address of case 11
                  .
                  .
                  .
           ¦
           ¦
      ___  ¦               ; address of case 19
           ¦
                  .
                  .
                  .
           ¦
      ___  ¦               ; address of case 23
           ¦
  <--------¦               ; first statement following END CASE
```

End Listing Nine

A Fast Circle Routine

In some instances it is necessary to draw partial circles or arcs and at other times complete circles are required. The algorithm presented here draws complete circles accurately and quickly and is relatively simple to program. The routine is written for the IBM Personal Computer using IBM's Macro Assembler, and can be implemented as an IBM Pascal procedure.

The Algorithm

Circle algorithms generally use a transformation from polar to rectangular coordinates by the following relationships:

$$X = R \cos(A)$$
$$Y = R \sin(A)$$

where the (X,Y) coordinates produce the locus of points for the given radius R as the angle A varies from 0 to 2 pi radians. This method requires trigonometric routines to compute sine and cosine values at least for the start and end points and for an angular increment.

A simpler method which does not require sine and cosine values is obtained by elementary calculus, which can be used to show that $dX/dY = -\text{tangent}(A) = -Y/X$. Also, $dY/dX = -\text{cotangent}(A) = -X/Y$. In other words, the change in X, given a small change in Y, is simply the negative of the ratio of Y to X. Similarly, the change in Y, given a small change in X, is simply the negative of the ratio of X to Y.

Given these relationships, the circle procedure is straightforward. For simplicity it will be assumed that the circle center is at the origin of our coordinate system. It is not difficult to adjust for a change in origin.

The first point plotted is chosen as Y=R and X=O, where R is the radius measured in screen column units. Y is incremented by one unit and X is decremented by Y/X units and the point plotted. This process is continued until Y/X=1. At that point (which corresponds to a 45 degree angle), we switch over to decreasing X by one unit and increasing Y by X/Y units. The process is continued until X/Y equals 0, i.e., when X is decremented to 0. At that point an arc from 0 to 90 degrees has been drawn. The complete circle can be obtained by using the symmetry of the circle to compute the corresponding points in each of the three remaining circle quadrants as we trace out the arc from 0 to 90 degrees.

The switch over when Y/X equals 1 is required because the tangent will approach infinity as the angle approaches 90 degrees, and overflow will occur somewhere prior to that point.

To actually implement the algorithm, we have to adjust for the origin located at the top left-hand corner of the video screen. Also we have to take into account the aspect ratio of the screen. When plotting 0 to 45 degrees, the X value must be multiplied by the inverse of the aspect ratio, and when plotting from 45 to 90 degrees, the Y value must be multiplied by the aspect ratio. Assuming a screen aspect ratio of 4/3, the aspect ratio to produce a circle in medium-resolution graphics is 5/6 and is 5/12 for high-resolution graphics.

The Assembly Routine

The routine incorporates the elements of the algorithm just described. It is also necessary to scale the computations by some factor to retain numerical accuracy, and then to rescale to obtain the points to plot. A factor of 1000 gives sufficient accuracy for the PC's high-resolution mode.

As mentioned, the circle routine can be implemented as a Pascal procedure. Listing 1 (page 35) is the circle routine and Listing 2 (page 37) is a simple calling routine which generates an internal calling sequence as used by IBM Pascal.

Pascal sets up a frame on the stack for an active procedure which is invoked by a FAR call. First, the calling program pushes the procedure arguments from left to right onto the stack. Then the caller's segment address and offset are pushed onto the stack. When the procedure receives control, it should first save the caller's base pointer by pushing BP onto the stack. Then the BP register is set equal to the SP register (stack pointer) and the procedure can then reference the arguments by adding six bytes to the BP register – four for the caller's return address and two for the caller's base pointer which has been pushed onto the stack. Thus, assuming integer value arguments, the first variable on the stack can be addressed by BP+6, the second by BP+8, and so on.

To return, the procedure must restore the caller's base pointer in BP and the SP register must be restored by a RET x, where x is the number of bytes on the stack occupied by the arguments which were passed.

Note that the IBM-supplied BIOS routine is used to write the points to screen. That is accomplished by storing 12 in the AH register, the X coordinate in the CX register, the Y coordinate in the DX register, and then issuing an INT 10H instruction. In this routine, the INT 10H instruction is defined as the MACRO, BIOSCALL, the expansion of which is denoted by "+" in the listing.

DDJ

(Listing begins on page 35)

by Daniel L. Lee

Daniel L. Lee, 1401 E. 55th Street, Apt. 601, Chicago, IL 60615.

Mr. Lee refuses to reserve any rights, commercial or otherwise, to this procedure.

Fast Circle Plot (Text begins on page 32)

Listing One

```
                        TITLE 'Fast Circle Plot'

                        BIOSCALL MACRO
                               INT 10H        ;BIOS service id in AH
                               ENDM

0000                    STACK SEGMENT PARA STACK 'STACK'
0000    40 [                   DB 64 DUP('STACK   ')
           53 54 41 43
           4B 20 20 20
                   ]

0200                    STACK ENDS


0000                    CSEG  SEGMENT PARA 'CODE'
                        ;-----------------------------------------
                        ;PROCEDURE  CIRCLE(X,Y,RADIUS,NUMER,DENOM,
                        ;                              COLOR:INTEGER)
                        ;Dan Lee  July 1, 1982
                        ;SourceWare
                        ;
                        ;draws a circle at center (x,y) with aspect
                        ;ratio numer/denom; radius in column units
                        ;
                        ;assumes entry via inter-segment call
                        ;
                        ;FRAME:   VALUE X     : BP+16
                        ;         VALUE Y     : BP+14
                        ;         VALUE RADIUS: BP+12
                        ;         VALUE NUMER : BP+10
                        ;         VALUE DENOM : BP+8
                        ;         VALUE COLOR : BP+6
                        ;-----------------------------------------
0000                    CIRCLE  PROC FAR
                               ASSUME CS:CSEG,SS:STACK
                               PUBLIC CIRCLE
0000  55                       PUSH BP         ;caller's frame pntr
004D  2B CB                    SUB CX,BX       ;get 2nd quad
004F  2B CB                    SUB CX,BX       ;X+origin
0051  50                       PUSH AX         ;save write dot parms
                               BIOSCALL        ;write 2nd quad point
0052  CD 10             +     INT 10H         ;BIOS service id in AH
0054  58                       POP AX          ;restore write dot parms
0055  03 D7                    ADD DX,DI       ;get 3rd quad
0057  03 D7                    ADD DX,DI       ;Y+origin
0059  50                       PUSH AX         ;save write dot parms
                               BIOSCALL        ;plot 3rd quad point
005A  CD 10             +     INT 10H         ;BIOS service id in AH
005C  58                       POP AX          ;restore write dot parms
005D  03 CB                    ADD CX,BX       ;get 4th quad
005F  03 CB                    ADD CX,BX       ;X+origin
                               BIOSCALL        ;plot 4th quad point
0061  CD 10             +     INT 10H         ;BIOS service id in AH
                        ;
                        ; cx now at original point
                        ;
0063  87 CB                    XCHG CX,BX      ;get 1st quad X
0065  47                       INC DI          ;get new Y
0066  8B C7                    MOV AX,DI       ;AX=Y
0068  8B 5E 08                 MOV BX,[BP+8]   ;BX=inv aspect*1000
006B  F7 EB                    IMUL BX         ;AX=Y*inv aspect*1000
006D  F7 F9                    IDIV CX         ;AX=TAN*inv aspect*1000
006F  33 D2                    XOR DX,DX       ;zero remainder
0071  8B F0                    MOV SI,AX       ;SI=TAN*inv aspect*1000
0073  F7 FB                    IDIV BX         ;AX=TAN
0075  3D 0001                  CMP AX,1        ;TAN=1?
0078  5A                       POP DX          ;DX=hi word X*1000
0079  58                       POP AX          ;AX=lo word X*1000
007A  73 0B                    JAE CR7         ;yes, go to next sector
007C  F7 DE                    NEG SI          ;to decrement X
007E  BB FFFF                  MOV BX,-1       ;negative carry
0081  03 C6                    ADD AX,SI       ;new X value
0083  13 D3                    ADC DX,BX       ;hi word carry
0085  EB A3                    JMP SHORT CR5   ;plot new point
```

```
0001  8B EC                    MOV BP,SP       ;set proc frame pntr
0003  8B 46 0A                 MOV AX,[BP+10] ;get aspect numer and
0006  BB 03E8                  MOV BX,1000     ;scale it by 1000
0009  F7 EB                    IMUL BX
000B  8B 4E 08                 MOV CX,[BP+8]  ;get aspect denom
000E  F7 F9                    IDIV CX         ;AX=aspect*1000
0010  50                       PUSH AX         ;store aspect*1000
0011  91                       XCHG AX,CX      ;get denom in AX
0012  8B 4E 0A                 MOV CX,[BP+10] ;get numer in CX
0015  F7 EB                    IMUL BX         ;AX=denom*1000
0017  F7 F9                    IDIV CX         ;AX=inv aspect*1000
0019  89 46 08                 MOV [BP+8],AX  ;store it
001C  58                       POP AX          ;get aspect*1000
001D  89 46 0A                 MOV [BP+10],AX ;and store it
                        ;
                        ; start by incrementing Y by one unit and
                        ; decrementing X by TAN units*inv aspect
                        ; start at (RADIUS,Y) and plot to 45 deg
                        ;
0020  8B 46 0C                 MOV AX,[BP+12] ;get radius for
0023  BB 03E8                  MOV BX,1000     ;initial X and scale
0026  F7 EB                    IMUL BX         ;up by 1000
0028  33 FF                    XOR DI,DI       ;zero initial Y value

002A  50                CR5:   PUSH AX         ;save lo word X*1000
002B  52                       PUSH DX         ;save hi word X*1000
002C  33 DB                    XOR BX,BX       ;begin round process
002E  05 01F4                  ADD AX,500      ;'one-half'
0031  13 D3                    ADC DX,BX
0033  BB 03E8                  MOV BX,1000     ;rescale X by 1000
0036  F7 FB                    IDIV BX         ;to graph
0038  8B D8                    MOV BX,AX       ;BX=1st quad X
003A  03 46 10                 ADD AX,[BP+16] ;add X origin
003D  8B 56 0E                 MOV DX,[BP+14] ;get Y origin
0040  2B D7                    SUB DX,DI       ;and sub Y to plot
0042  8B C8                    MOV CX,AX       ;get X to plot
0044  8A 46 06                 MOV AL,[BP+6]  ;get color
0047  B4 0C                    MOV AH,12       ;write dot funct select
0049  50                       PUSH AX         ;save write dot parms
                               BIOSCALL        ;write 1st quad point
004A  CD 10             +      INT 10H         ;BIOS service id in AH
004C  58                       POP AX          ;restore write dot parms
```

(Continued on page 35, column 2)

```
                        ;
                        ; plot 45 to 90 degrees
                        ; now decrease X by one unit and
                        ; increase Y by COT units*aspect ratio
                        ;
0087  8B C7             CR7:   MOV AX,DI       ;get next Y to plot and
0089  BB 03E8                  MOV BX,1000     ;scale by 1000
008C  F7 EB                    IMUL BX         ;DX:AX=Y*1000
008E  8B F9                    MOV DI,CX       ;DI=last X value
0090  4F                       DEC DI          ;next X to plot

0091  50                CR8:   PUSH AX         ;save lo word Y*1000
0092  52                       PUSH DX         ;save hi word Y*1000
0093  33 DB                    XOR BX,BX       ;begin round process
0095  05 01F4                  ADD AX,500      ;'one-half'
0098  13 D3                    ADC DX,BX
009A  BB 03E8                  MOV BX,1000     ;rescale Y to plot
009D  F7 FB                    IDIV BX         ;AX=Y
009F  8B D8                    MOV BX,AX       ;BX=1st quad Y coord
00A1  03 46 0E                 ADD AX,[BP+14] ;add Y origin
00A4  8B 4E 10                 MOV CX,[BP+16] ;CX=X origin
00A7  03 CF                    ADD CX,DI       ;X to plot
00A9  8B D0                    MOV DX,AX       ;Y to plot
00AB  8A 46 06                 MOV AL,[BP+6]  ;get color
00AE  B4 0C                    MOV AH,12       ;write dot funct select
00B0  50                       PUSH AX         ;save write dot parms
                               BIOSCALL        ;write 1st quad point
00B1  CD 10             +      INT 10H         ;BIOS service id in AH
00B3  58                       POP AX          ;restore write dot parms

00B4  2B CF                    SUB CX,DI       ;get 2nd quad
00B6  2B CF                    SUB CX,DI       ;X to plot
00B8  50                       PUSH AX         ;save write dot parms
                               BIOSCALL        ;plot 2nd quad point
00B9  CD 10             +      INT 10H         ;BIOS service id in AH
00BB  58                       POP AX          ;restore write dot parms
00BC  2B D3                    SUB DX,BX       ;get 3rd quad
00BE  2B D3                    SUB DX,BX       ;Y to plot
00C0  50                       PUSH AX         ;save write dot parms
                               BIOSCALL        ;write 3rd quad point
00C1  CD 10             +      INT 10H         ;BIOS service id in AH
```

(Continued on next page)

Fast Circle Plot

Listing One (Listing One continued, text begins on page 32)

```
00C3  58                        POP AX          ;restore write dot parms
00C4  03 CF                     ADD CX,DI       ;get 4th quad
00C6  03 CF                     ADD CX,DI       ;X to plot
                                BIOSCALL        ;plot 4th quad point
00C8  CD 10                +    INT 10H         ;BIOS service id in AH
00CA  2B 56 0E                  SUB DX,[BP+14] ;DX=Y-Y origin
00CD  F7 DA                     NEG DX          ;Y origin adjust
00CF  87 CA                     XCHG CX,DX      ;CX=Y
00D1  0B FF                     OR DI,DI        ;90 deg?
00D3  78 1D                     JS CR11         ;yes,exit
00D5  4F                        DEC DI          ;get new X
00D6  8B C7                     MOV AX,DI       ;AX=X
00D8  8B 5E 0A                  MOV BX,[BP+10] ;BX=aspect*1000
00DB  F7 EB                     IMUL BX         ;AX=aspect*1000*X
00DD  F7 F9                     IDIV CX         ;AX=aspect*1000*COT
00DF  8B F0                     MOV SI,AX       ;SI=change in Y


00E1  5A                        POP DX          ;DX=hi word Y*1000
00E2  58                        POP AX          ;AX=lo word Y*1000
00E3  33 DB                     XOR BX,BX
00E5  0B F6                     OR SI,SI        ;for sign check
00E7  79 03                     JNS CR10        ;positive
00E9  BB FFFF                   MOV BX,-1       ;negative carry
00EC  03 C6               CR10: ADD AX,SI       ;AX=new X value
00EE  13 D3                     ADC DX,BX       ;hi word carry
00F0  EB 9F                     JMP SHORT CR8   ;plot next point
                          ;
                          ; exit
                          ;



00F2  83 C4 04            CR11: ADD SP,4        ;adjust stack pntr
00F5  5D                        POP BP          ;caller's frame pntr
00F6  CA 000C                   RET 12          ;release parms
00F9                      CIRCLE ENDP
00F9                      CSEG  ENDS
                                END
```

End Listing One

Listing Two

```
                          TITLE CALLER - CALLS CIRCLE SUBROUTINE

                                EXTRN CIRCLE:FAR;external subroutine

0000                      STACK SEGMENT PARA STACK 'STACK'
0000    40 [                    DB  64 DUP('STACK   ')
            53 54 41 43
            4B 20 20 20
                          ]

0200                      STACK ENDS


0000                      CALLER SEGMENT PARA 'CODE'
0000                      START PROC FAR
                                ASSUME CS:CALLER,SS:STACK
0000  1E                       PUSH DS    ;save ret seg on stack
0001  33 C0                    XOR AX,AX  ;zero for ret offset
0003  50                       PUSH AX    ;and save on stack
0004  B0 04                    MOV AL,4   ;color/graphics select
0006  CD 10                    INT 10H    ;set mode
0008  B8 00A0                  MOV AX,160 ;X origin coord
000B  50                       PUSH AX    ;save on stack
000C  B8 0064                  MOV AX,100 ;Y origin coord
000F  50                       PUSH AX    ;save on stack
0010  B8 0032                  MOV AX,50  ;circle radius
0013  50                       PUSH AX    ;save on stack
0014  B8 0005                  MOV AX,5   ;aspect numer
0017  50                       PUSH AX    ;save on stack
0018  B8 0006                  MOV AX,6   ;aspec denom
001B  50                       PUSH AX    ;save on stack
001C  B8 0001                  MOV AX,1   ;foregrnd color
001F  50                       PUSH AX    ;save on stack
0020  9A 0000 ---- E           CALL CIRCLE;draw circle
0025  CB                       RET        ;far ret to DOS
0026                      START ENDP
0026                      CALLER ENDS
                                END
```

End Listing Two

Enhancing the C Screen Editor

by Alan D. Howard

Alan D. Howard, Route 3, Box 680, Crozet, VA 22932.

In the January 1982 issue of Dr. Dobb's *we presented Edward Ream's popular Small-C screen editor. Readers received a fine editor which, at the same time, was an excellent candidate for extension and modification. Alan Howard has provided us with an extensive set of enhancements. Those who wish to implement changes in the original editor should find guidance and inspiration from Mr. Howard's work.*

Mr. Howard has chosen a modular approach to the problem and his changes should be easy to implement. Mr. Ream himself has attacked the situation by significant revision of the entire editor. We will be publishing Mr. Ream's revision, called RED, in the near future. Comparison of these two approaches should provide insight into programming techniques and implementation tradeoffs, and we feel fortunate to be able to publish both pieces of work.

While the January and May 1982 back issues are no longer available (the bound volume containing them should be out later this year), we understand the original editor may still be obtained from a variety of sources, including several computer bulletin boards. Edward Ream still distributes it himself for $50 on 8-inch, CP/M diskettes, and can be reached at 1850 Summit Avenue, Madison, WI 53705. A BDS C version may be obtained from The C User's Group, Box 287, Yates Center, KS 66783 (which sells it for $8 in a variety of formats) and Steve Passe's C node computer bulletin board (contact the C Users' group for details).

At the beginning of Edward Ream's article introducing his Small-C Screen Editor in *Dr. Dobb's Journal* (No. 63, January 1982), he asked whether our present editor lacked flexibility, transportability, and extensibility. Although my mainstay editor has been PIE from Software Toolworks, I have often been annoyed at its inability to edit large files or to make major structural changes in text. On the other hand, Ream's editor as presented did not go much beyond PIE and seemed clumsy in its Edit mode. It did, however, encourage adapting it to fit individual needs since it was presented in source code. Thus, after several nights of typing and several more nights of debugging, I had the C editor up and running (those of you who want to join me doing it the hard way, be sure to note the bug fixes in the May issue, No. 67). I have been modifying the editor during the last few months and it has evolved to the point that I feel some of *Dr. Dobb's* readers may be interested. The changes are of several types:

(1) Extensions to the file-handling and buffer management to permit editing large files, extracting parts of the text to the write file, and moving and copying portions of the text within the file.

(2) Changing the edit mode operation to recognize special keys on the H19 Terminal or H89 Computer instead of single-letter commands. As a bonus, the editor now assumes *eXchange* mode for any other typed key. The code can be easily modified for other terminal protocols.

(3) Addition of a few new commands to the edit-mode and some changes in rules of operation.

(4) Slight modifications to allow modular compilation using the Software Toolworks C/80 compiler.

(5) Enhancements to improve the speed of response in edit mode and during search and replace operations.

These enhancements are discussed below as explanations for the listing that accompanies this article. Only those variables, declarations, and functions that have been changed from the original listing are listed. A documentation file is included in the listing which summarizes the expanded features and all commands.

Changes to File Handling and Command-Mode Operation

Most of the changes to the command-mode are concerned with adding flexibility to file handling and major buffer alterations. Separate read and write files are now maintained, with *load* <filename> specifying the read file name, clearing the buffer, reading the file, and closing the read file if it all fits in the buffer. Otherwise as much as fits is read in, and the read file remains open. If a filename is specified in the command line when the editor is loaded, the filename is automatically passed to the *load* command. The *open* <filename> command opens the read file without clearing the buffer or reading anything in. More from the read file can be added to the buffer with either the *rest* or *read* <n> commands. *Rest* reads in as much more as possible and gives the option of clearing the buffer, but *read* <n> reads only <n> more lines without clearing the buffer. The write file is specified by *name* <filename> for a new file, or *delname* <filename> for writing over an existing file (this replaces the *resave* command). *Rename* <filename> closes the existing write file and opens a new one. *Write* <n> writes <n> lines from the buffer to the write file, deleting those lines. To balance the existing *append* <filename> command, an *extract* <from to> command has been added to write the indicated line range to the write file without deleting those lines from the buffer. Finally, to allow flexibility, the read and write files can be closed by *closeread* and *closewrite*.

Major structural changes in the file can now be made using the *copy* <from to n> and *move* <from to n> commands, which take n lines from <from> and copy them to before line <to>. In addition, *move* deletes the <n> lines at <from>. For speed, these routines open up the new space, and then do the copying or moving. No move or copy is allowed if there is not room in the buffer. If buffer space is tight, or if the move or copy is to a part of the file not currently in memory, then use the *extract* and *append* commands, which use the disk as a buffer.

Because tabs are treated as ordinary characters, *showtab* has been included to allow tabs to be shown in reverse video. This mode is reset by *hidetab* (the default mode) where tabs are just blanks.

The following command-mode commands are unaltered from the original: *append, change, clear, delete, dos, find, g, list, search,* and *tabs.*

Changes to Edit and Insert Modes

The most important change to the edit mode is the replacement of the single-letter commands by escape-character sequences generated by the special keys on the H19/H89 keyboard. Similar keys, but with different escape sequences, are found on many other terminals and computers. This allows default use of the *eXchange* mode (replacing contents of cursor position with the typed keystroke) for normal keys, including the space bar. The *eXchange* mode has been speeded up considerably to allow rapid typing. Most special characters and edit mode commands retain their original functions (see the documentation portion of the listing

for details), but two new commands have been created for the edit mode: the *HOME* key moves alternately to the top and bottom line of the screen, and the *ERASE* key erases from the current cursor position to the end of the line. More subtle changes have also been made. The *RETURN* key moves to the beginning of the next line in edit mode, but acts as the *insert down* key in the insert mode. Both the *DC* and *DELETE* keys delete the character at the cursor, but *BACKSPACE* deletes the character to the left of the cursor, the same as the original *delete character* special key. The only commands remaining as control codes are *split* and *join*. There is some room for expansion: three keys are currently assigned to force command mode, but could be reassigned. Note that the *ESC* key must be pressed twice to be recognized, because it is also issued by the special character codes of the keyboard. My choice of key assignments and features has admittedly been influenced by the PIE editor.

Implementation Details

I'd like to start out with an unsolicited endorsement of the Software Toolworks C/80 Compiler, now in version 2.0 and incorporating almost all of the C language. It's a descendent of Small-C and is inexpensive, fast, and powerful. Its salient feature in the context of the editor is support for modular compilation using Microsoft M80 and L80. Since the editor is broken into nine fairly independent modules, changes can be made in one without having to recompile everything. Also, SID can be used for debugging using global variable and function names. The disadvantage is that variable and function names must be distinct at six characters' length. This has necessitated renaming several functions in the editor, as listed in Figure 1 (page 42). (I have not listed each occurrence of the altered names in the listing; I leave that as a test of the capabilities of your present editor.) Also, to force the main buffer to be at the end of the program, the short program MBUFFER.MAC must be assembled by M80 and be the *last* module linked (after the C/80 CLIBRARY module).

Most details of the changes are explained in the listing on page 43. The functions that are completely unchanged are not included, but a note indicates their position. In general, I have listed the entire function if there has been any change.

Concluding Remarks

This editor is a very useful addition to my stable of inexpensive editors, and is particularly suited to editing large files, interleaving text from other files, and breaking up a file or portions of a file into smaller pieces. It doesn't do everything, and reading and writing to files are considerably slower than for editors written in assembly language. I suggest those with H89's go through the bothersome task of reassembling their BIOS with the type-ahead buffer option so that rapid typing will not lose characters.

When I move on to my next system, I will feel secure that I have its first editor waiting in the wings, and it won't cost me twice as much as my previous editor for the same features. The modifications to the editor are in the public domain, and a somewhat earlier version has been submitted to the SIG/M User Group (Amateur Computer Group of New Jersey, Box 97, Iselin, NJ 08830).

(Figure 1 on page 42)

(Listing begins on page 43)

Figure 1. (Text begins on page 38)

Altered Function Names

Original	New	Original	New
search1	suurch	pmtcol1	pmt1col
outgetx	outxget	sysmovdn	sysdnmov
outgety	outyget	sysmovup	sysupmov
outhasdn	outdnhas	bufdeln	bufndel
outhasup	outuphas	bufout1n	buf1nout
pmtmode1	pmt1mode	bufmovup	bufupmov
pmtfile1	pmt1file	bufmovdn	bufdnmov
pmtline1	pmt1line	buffer	mbuffer

DOCUMENTATION FOR "C" EDITOR – FEBRUARY 28, 1983

CP/M-H19-H89 Version; written by Edward K. Ream; *Dr. Dobb's Journal*, January 1982, V. 7, Issue 1; modified by Alan D. Howard.

COMMAND MODE COMMANDS: (May be entered in upper or lower case)

Command	Arguments	Function
append	<filename>	Insert the named file into buffer at cursor
change	<from to>	Make indicated changes in lines in range <from to>
clear		Erase the buffer
closeread		Close the read file
closewrite		Close the write file
copy	<from to n>	Copy <n> lines from <from> to before <to>
delete	<from to>	Delete all lines in the range <from to>
delname	<filename>	Make the indicated file the write file; no error if file already exists
dos		Exit from editor to operating system
extract	<from to>	Write the designated lines to write file
find		Search for a pattern; enter edit mode
g	<n>	Go to line <n> and enter edit mode
help		List the command mode commands to the screen
hidetab		Do not show tabs in reverse video (default)
list	<from to >	List the indicated lines to printer
load	<filename>	Open the indicated file as the read file; clear the buffer; load the buffer from the file; close the read file if entire file read
move	<from to n>	Move <n> lines from <from> to before <to>
name	<filename>	Make the indicated file the write file; error if file already exists
open	<filename>	Open the indicated file as the read file
read	<n>	Read <n> lines from read file to end of buffer; close the read file if entire file read
rename	<filename>	Close the write file; open new write file
rest		Clear the buffer if requested; load the buffer from read file; close the read file if entire file read
save		Save the buffer into the write file; buffer is unchanged
search	<from to>	Print all lines that match a pattern
showtab		Show tabs in reverse video
tabs	<n>	Set tab stops at every <n> columns
write	<n>	Write <n> lines from front of buffer to the write file; the lines are deleted from the buffer
(blue key)		Enter edit mode
(IC key)		Enter insert mode

EDIT AND INSERT MODE KEY COMMANDS:
(Commands marked † not valid in insert mode; Commands marked + force edit mode)

Key	Action
(f1)	† Scroll down until any key pressed
(f2)	† Move to end of line
(f3) <n> CR	† Go to line <n>
(f4)	† Move to beginning of line
(f5)	† Scroll up until any key pressed
(ERASE)	† Erase from cursor to end of line
(Blue)	Enter edit mode
(Red)	Abort changes to current line
(White)	Enter command mode
(BACKSPACE)	Delete character before cursor
(DELETE)	Delete character at cursor; same as (BACKSPACE) at end of line
(RETURN)	Insert mode: add line below current line, move to the new line Edit mode: move to next line
(LINE FEED)	Insert line at cursor: move current line down, enter insert mode
(IC)	Enter insert mode
(DC)	Delete character at cursor; same as (BACKSPACE) at end of line
(IL)	Insert line at cursor: move current line down, enter insert mode
(DL)	Delete current line
(0) <char>	† (Zero on auxiliary keypad); search for character and move cursor to character
(.) <char>	† (Period on auxiliary keypad); delete characters from cursor up to but not including <char>
(ENTER)	Enter command mode
(HOME)	† Move cursor alternately to top and bottom of screen
(Control-S)	Split line at cursor
(Control-P)	Join current line with line above (if room)
(ESC ESC)	Enter command mode
(Up-arrow)	+ Up one line
(Down-arrow)	+ Down one line
(Left-arrow)	Left one character
(Right-arrow)	Right one character
(All others)	Any other printing key (and TAB key) act as follows: Edit mode: replace cursor with character; Insert mode: insert character

End Figure 1.

(Continued on page 42, column 2)

(Continued on page 43, column 1)

Screen Editor Enhancements

(Text begins on page 38)

```
/* ED0.C */

/* INCLUSION FILE */

#define MAXLEN  80      /* 80 character video screen */
#define MAXLEN1 81      /* MAXLEN+1 */
#define SYSFNMAX 15
#define EOS     0
#define OK      1
#define ERR     -1
#define EOF     -2
#define YES     1
#define NO      0
#define CR      13
#define LF      10
#define TAB     9
#define HUGE    32000
```

```
/* ED1.CCC */

/* INCLUSION FILE */

#define UP1     10
#define DOWN1   13
#define UP2     21
#define DOWN2   4
#define LEFT1   25      /* new code assignment */
#define RIGHT1  18
#define INS1    14
#define EDIT1   5
#define ESC1    27
#define DEL1    127
#define ZAP1    26
#define ABT1    24
#define SPLT1   19
#define JOIN1   16
#define DTOCH   1       /* kill mode in edit */
#define GTOCH   2       /* search mode in edit */
#define HOME    6       /* home cursor or to bottom of screen */
#define DSCROL  7       /* scroll down */
#define GOTO    17      /* go to line */
#define LSTRT   11      /* go to beginning of line */
#define LEND    12      /* go to end of line */
#define USCROL  15      /* scroll up */
#define ERASE   20      /* erase to end of line */
#define SCRNW   81      /* H19 screen width */
#define SCRNW1  80      /* SCRNW-1 */
#define SCRNL   24      /* H19 screen length */
#define SCRNL1  23      /* SCRNL-1 */
#define SCRNL2  22      /* SCRNL-2 */
#define SCRNL3  13      /* half a screen length */
#define LISTW   80      /* width of listing */
#define LFTDEL   8      /* delete to left of cursor */
```

```
/* ED2.C */

#include ed0.c
#include ed1.ccc
#define SIGNON "E.K. Ream/Dr Dobb's Editor - H89/H19 enhancement: Feb. 20, 1983"
#define HELP1 "open <f/n> | name <f/n> | rename <f/n> | delname <f/n>"
#define HELP2 "load <f/n> | append <f/n> | rest | read <n> | write <n> | save"
#define HELP3 "extract <f,t> | closeread | closewrite"
```

```
        eddn();         /* enter insert mode but */
        pmtline();      /* just advances to next line */
        edbegin();
        pmtcol();
}
else if (c==RIGHT1) {   /* go to end of line */
        edright();
        pmtcol();
}
else if (c==ERASE) {    /* erase to end of line */
        ederase();
        pmtcol();
}
else if (c==HOME) {     /* move cursor alternatingly to */
        edhome();       /* top and bottom of screen */
        pmtline();
}
else if (c==LSTRT) {    /* move to beginning of line */
        edbegin();
        pmtcol();
}
else if (c==DSCROL) {   /* scroll down */
        pmtmode("edit: scroll");
        while (bufnrbot()==NO) {
                if (chkkey()==YES) {
                        break;
                }
                if (eddn()==ERR) {
                        break;
                }
        }
        pmtedit();
}
else if (c==LEND){      /* move to end of line */
        edend();
        pmtcol();
}
else if (c==GOTO){      /* go to line entered */
        x=outxget();
        y=outyget();
        pmtcmnd("edit: goto: ",sbuffer);
        if(number(sbuffer,&v)) {
                edgo(v,0);
        }
        else {
                outxy(x,y);
        }
        pmtedit();
}
else if (c==DTOCH){     /* kill to character entered */
        pmtmode("edit:  kill");
        c=syscin();
        if ((special(c)==NO) &
            (control(c)==NO)) {
                edkill(c);
        }
        pmtedit();
}
else if (c==GTOCH){     /* search to character entered */
        pmtmode("edit: search");
        c=syscin();
        if ((special(c)==NO) &
            (control(c)==NO)) {
                edsrch(c);
        }
        pmtedit();
```

```
#define HELP4 "clear | delete <f t> | move <f t n> | copy <f t n>"
#define HELP5 "find | search <f t> | change <f t>"
#define HELP6 "tabs <n> | showtab | hidetab | help | dos | g <n>"
#define CMNDMODE        1
#define INSMODE         2
#define EDITMODE        3
#define EXITMODE        4
main(argc,argv) char *argv[];
{
int mode;
int i;
        syscout(ESC1); /*alternate keyboard mode*/
        syscout('=');
        fmtassn(NO);
        fmtset(8);
        fmtntab();
        outclr();
        outxy(0,SCRNL1);
        message(SIGNON);
        outxy(0,1);
        fileclear();    /* make sure no read or write files indicated */
        bufnew();
        mode=CMNDMODE;
        if (argc>1) {
                open(argv[1],NO);
                rest(argv[0]);
                outxy(0,1);
        }
        edgetln();
        while(1){
                if (mode ==EXITMODE) {
                        break;
                }
                else if (mode==CMNDMODE) {
                        mode=command();
                }
                else if (mode==EDITMODE) {
                        mode=edit();
                }
                else if (mode==INSMODE) {
                        mode=insert();
                }
                else {
                        syserr("main: no mode");
                        mode=EDITMODE;
                }
        }
}
edit()
{
char sbuffer[SCRNW1];
int v;
int x,y;
char c;
        fmtedit();
        while(1){
                c=syscin();     /* tolower eliminated */
                if (c==ESC1) {          /* enter command mode */
                        return(CMNDMODE);
                }
                else if (c==INS1) {     /* enter insert mode */
                        return(INSMODE);
                }
                else if (special(c)==YES) {
                        if (c==UP1){    /* DOWN1 now treated separately */
                                return(INSMODE);
                        }
                        else {
                                continue;
                        }
                }
                else if (c==DOWN1) {    /* DOWN1 now does not    */
                }
                else if (c==USCROL) {   /* scroll up */
                        fmtmode("edit: scroll");
                        while (bufattop()==NO) {
                                if (chkkey()==YES) {
                                        break;
                                }
                                if (edup()==ERR) {
                                        break;
                                }
                        }
                        fmtedit();
                }
                else {          /* editor now exchanges any other character */
                        if ((special(c)==NO) &  /* with cursor */
                            (control(c)==NO)) {
                                edchng(c);
                        }
                        fmtcol();  /* only need to update column number */
                }
        }
}
insert()
{
char c;
        fmtmode("insert");
        while(1) {
                c=syscin();
                if (c==ESC1) {
                        return(CMNDMODE);
                }
                else if (c==EDIT1) {
                        return(EDITMODE);
                }
                else if (c==INS1) {
                        ;
                }
                else if (c==DOWN1) {    /* in insert mode DOWN1 treated  */
                        ednewdn();      /* differently from edit mode    */
                        fmtline();      /* inserts new line */
                        continue;
                }
                else if (special(c)==YES) {
                        if ((c==UP2)|(c==DOWN2)) {
                                return(EDITMODE);
                        }
                        else {
                                continue;
                        }
                }
                else if (control(c)==YES) {
                        continue;
                }
                else {
                        edins(c);
                        fmtcol();
                }
        }
}

/***** control(c) unchanged *****/

special(c) char c;
{
int k;
        if (c==JOIN1) {
                edjoin();
                fmtline();
                return(YES);
        }
        if (c==SPLT1) {
```

(Continued on page 43, column 2)

(Continued on next page)

Screen Editor Enhancements

(Listing continued, text begins on page 38)

```
                edsplit();
                pmtline();
                return(YES);
        }
        if (c==ABT1) {
                edabt();
                pmtcol();
                return(YES);
        }
        else if (c==LFTDEL) {           /* delete to left of cursor */
                edldel();
                pmtcol();
                return(YES);
        }
        else if (c==DEL1) {             /* delete at cursor */
                edcdel();
                pmtcol();
                return(YES);
        }
        else if (c==ZAP1) {
                edzap();
                pmtline();
                return(YES);
        }
        else if (c==UP2) {
                edup();
                pmtline();
                return(YES);
        }
        else if (c==UP1) {
                ednewup();
                pmtline();
                return(YES);
        }

/* DOWN1 (CR) is no longer a special character */

        else if (c==DOWN2) {
                eddn();
                pmtline();
                return(YES);
        }
        else if (c==LEFT1) {
                edleft();
                pmtcol();
                return(YES);
        }
        else if (c==RIGHT1) {
                edright();
                pmtcol();
                return(YES);
        }
        else  {
                return(NO);
        }
}
command()
{
int v;
char c;
char args[SCRNW1];
char *argp;
int topline;
int ypos;
int oldline;
int k;
        edrepl();
        oldline=buf1n();
        ypos=outyget();
}
else if (lookup(args,"open")) {          /* new */
        open(args,YES);
}
else if (lookup(args,"load")) {          /* changed */
        open(args,YES);
        rest(args);
}
else if (lookup(args,"name")) {
        name(args);
}
else if (lookup(args,"write")) {         /* new */
        writel(args);
}
else if (lookup(args,"rename")) {        /* new */
        rename(args);
}
else if (lookup(args,"delname")) {       /* new */
        delname(args);
}
else if (lookup(args,"closewrite")) {    /* new */
        closewrite(args);
}
else if (lookup(args,"showtab")) {     /* new */
        fmtytab();
}
else if (lookup(args,"hidetab")) {     /* new */
        fmtntab();
}
else if (lookup(args,"closeread")) {     /* new */
        closeread(args);
}
else if (lookup(args,"help")) {          /* new */
        outclr();
        outxy(0,SCRNL1);
        message(HELP1);
        message(HELP2);
        message(HELP3);
        message(HELP4);
        message(HELP5);
        message(HELP6);
}
else if (lookup(args,"save")) {
        save(args);
}
else if (lookup(args,"read")) {           /* new */
        getit(args);
}
else if (lookup(args,"move")) {          /* new */
        moveit(args);
}
else if (lookup(args,"copy")) {          /* new */
        copyit(args);
}
else if (lookup(args,"search")) {
        search(args);
}
else if (lookup(args,"tabs")) {
        tabs(args);
}
else if (lookup(args,"rest")) {          /* new */
        rest(args);
}
else if (lookup(args,"extract")) {       /* new */
        extract(args);
}
```

```
        topline=oldline-ypos+1;
        while(1) {
                outxy(0,SCRNL1);
                fmtcrlf();
                fmtmode("command:");
                setcmnd(args,0);
                fmtcrlf();
                fmtline();
                c=args[0];
                if ((c==EDIT1)||(c==INS1)) {
                        if (oldline==bufln()) {
                                edgetln();
                                bufout(topline,1,SCRNL1);
                                outxy(0,ypos);
                        }
                        else {
                                edgo(bufln(),0);
                        }
        if (c==EDIT1) {
                return(EDITMODE);
        }
        else {
                return(INSMODE);
        }
}
else if (tolower(args[0])=='g'){
        argp=skipbl(args+1);
        if (argp[0]==EOS) {
                edgo(oldline,0);
                return(EDITMODE);
        }
        else if (number(argp,&v)==YES) {
                edgo(v,0);
                return(EDITMODE);
        }
        else {
                message("bad line number");
        }
}
else if (lookup(args,"append")) {
        append(args);
}
else if (lookup(args,"change")) {
        change(args);
}
else if (lookup(args,"clear")) {
        clear();
}
else if (lookup(args,"delete")) {
        delete(args);
}
else if (lookup(args,"dos")) {
        if (chkbuf()==YES) {
                closewrite();   /* write file closed on exit */
                syscout(ESC1); /*exit alternate keyboard mode*/
                syscout('>');
                syscout(ESC1); /*clear display*/
                syscout('E');
                return(EXITMODE);
        }
}
else if (lookup(args,"find")) {
        if ((k=find()) >= 0) {
                edgo(bufln(),k);
                return(EDITMODE);
        }
        else {
                bufgo(oldline);
                edgetln();
                message("pattern not found");
        }
}
else if (lookup(args,"list")) {
        list(args);
```

(Continued on page 45, column 2)

```
                else {
                        message("command not found");
                }
        }
}

/***** lookup() and setcmnd() unchanged *****/

-------------------------------------------------------------------------------

/* ED3.C */

#include ed0.c
#include ed1.ccc
int readfile = -1;      /* separate read and write files */
int writefile = -1;     /* -1 for inactive, file channel no. for active */
char rfilename[SYSFNMAX];       /* file names */
char wfilename[SYSFNMAX];
char databuf[MAXLEN1];          /* single buffer replaces redundant buffers */
fileclear()             /* initializes filenames to zero length */
{
        rfilename[0] = EOS;
        wfilename[0] = EOS;
}

/***** append() unchanged *****/

/***** the only alterations to change() are in amatch() references
       1st reference was: if(amatch(oldline,oldpat+1,0)==YES) {
                  is now: if (amatch(oldline,oldpat+1,NO)==0) {
       2nd reference was: if (amatch(oldline,oldpat,col++)==YES) {
                  is now: if ((col=amatch(oldline,oldpat,YES))>=0) {
*****/

clear()
{
        if (chkbuf()==YES) {
                outclr();
                outxy(0,SCRNL1);
                bufnew();
                message("buffer cleared");
                return(YES);
        }
        else{
                return(NO);
        }
}

/***** delete() unchanged *****/

find()
{
        return(search(bufln()+1,HUGE,YES));
}

/***** list() unchanged *****/

extract(args) char *args;       /* write indicated line range to writefile */
{
int oldline, from, to, n;
        if (writefile == -1) {
                message("file not opened");
                return;
        }
        oldline = bufln();
        if (get2args(args,&from,&to)==ERR) {    /* from and to are line range */
                return;
        }
        if (bufgo(from)==ERR){
                return;
```

(Continued on next page)

Screen Editor Enhancements

(Listing continued, text begins on page 38)

```
        }
        while ((bufatbot()==NO)&(bufln()<=to)) {        /* do the writing */
                n=bufgetln(databuf,MAXLEN);
                n=min(n,MAXLEN);
                if (pushline(writefile,databuf,n)==ERR){
                        break;
                }
                if (bufdn()==ERR) {
                        break;
                }
        }
        bufgo(oldline);
}
open (args,flag) char *args; int flag;
{
/* open a file for reading */
char locfn [SYSFNMAX];
int n;
int file;
int topline;
        if (readfile > 0) {             /* check for open readfile */
                message("read file still open");
                return;
        }
        if (flag==YES) {
                if (name1(args,locfn)==ERR) {
                        return;
                }
        }
        else  {
                if (name3(args,locfn)==ERR) {
                        return;
                }
        }
        if (locfn[0]==EOS) {
                message("no file argument");
                return;
        }
        if (chkbuf()==NO) {
                return;
        }
        if ((file=sysopen(locfn,"r"))==ERR) {
                message("file not found");
                return;
        }
        syscopfn(locfn, rfilename);     /* make the file name the readfile */
        readfile = file;
        pmtrfile(rfilename);
        bufnew();
}
setit (args) char *args;        /* new: add n lines to buffer if room */
{
int n;
int topline;
int nlines,npoint;
        if (readfile == -1) {
                message ("no read file");
                return;
        }
        pmtrfile(rfilename);
        if (setarg(args,&nlines)==ERR) {
                return;
        }
        if (nlines<1) {
                return;
        }
        if (bufgo(HUGE)==ERR) {
                return;
        }
        if (buflins(databuf,n)==ERR) {
                bufgo(1);
                topline=max(1,bufln()-SCRNL2);
                bufout(topline,2,SCRNL2);
                bufgo(topline);
                return;
        }
        if (bufdn()==ERR) {
                break;
        }
        }
        sysclose(readfile);     /* close file if all read */
        readfile = -1;
        rfilename[0] = EOS;
        pmtrfile(rfilename);
        bufgo(1);
        topline=max(1,bufln()-SCRNL2);
        bufout(topline,2,SCRNL2);
        bufgo(topline);
}
rename(args) char *args;        /* new: change name of writefile */
{
        if (writefile == -1) {
                message("no write file - call name");
                return;
        }
        sysclose(writefile);
        writefile = -1;
        wfilename[0]=EOS;
        pmtwfile(wfilename);
        if (name0(args,wfilename)==ERR) {
                return;
        }
        pmtwfile(wfilename);
}
name(args) char *args;          /* altered definition: name the writefile */
{
        if (writefile > 0) {
                message("write file open");
                return;
        }
        if (name0(args,wfilename)==ERR){
                return;
        }
        pmtwfile(wfilename);
}
name2(args,wfilename) char *args, *wfilename;   /* names new writefile */
{
int file;
        args=skiparg(args);
        args=skipbl(args);
        if (syschkfn(args)==ERR) {
                return(ERR);
        }
        syscopfn(args,wfilename);
        if ((file=sysopen(wfilename,"r"))!=ERR) {
                sysclose(file);
                message("disk file exists");
                return(ERR);
        }
        if ((file=sysopen(wfilename,"w"))==ERR) {
                return(ERR);
        }
        writefile = file;
        return(OK);
}
delname(args) char *args;       /* new: similar to old resave command, */
{                               /* but doesn't automatically write file */
        if (writefile > 0) {
                message("write file open");
                return;
        }
        if (name2(args,wfilename)==ERR){
```

```
        }
        npoint=1;
        while (npoint<=nlines) {        /* add the lines */
                npoint++;
                if ((n=readline(readfile,databuf,MAXLEN))>=0) {


                        if (n>MAXLEN) {
                                message("line truncated");
                                n=MAXLEN;
                        }
                        if (buflins(databuf,n)==ERR) {
                                bufgo(1);
                                topline=max(1,bufln()-SCRNL2);
                                bufout(topline,2,SCRNL2);
                                bufgo(topline);
                                return;
                        }
                        if (bufdn()==ERR) {
                                break;
                        }
                }
                else {
                        npoint = HUGE;      /* reached end of file - close it */
                        sysclose(readfile);
                        readfile = -1;
                        rfilename[0] = EOS;
                        fmtrfile(rfilename);
                }
        }
        bufgo(1);
        topline=max(1,bufln()-SCRNL2);
        bufout(topline,2,SCRNL2);
        bufgo(topline);
}
rest (args) char *args; /* new: try to read rest of readfile into buffer */
{
int n;
int topline;
        if (readfile == -1) {
                message ("no read file");
                return;
        }
        fmtrfile(rfilename);
/* if buffer has been changed and not saved, give option of clearing, */
/* otherwise add to end of buffer */
        if(bufchng()==YES) {
                fmtsout("buffer not saved, clear? ",0);
                fmtline();
                if (tolower(syscout(syscin()))=='y'){
                        outclr();
                        outxy(0,SCRNL1);
                        bufnew();
                        message("buffer cleared");
                }
                else {
                        bufgo(HUGE);
                }
        }
        else{
                outclr();
                outxy(0,SCRNL1);
                bufnew();
                message("buffer cleared");

        }
        while ((n=readline(readfile,databuf,MAXLEN))>=0) {
                if (n>MAXLEN) {
                        message("line truncated");
                        n=MAXLEN;
```

(Continued on page 47, column 2)

```
                        return;
                }
                fmtwfile(wfilename);
}
name2(args,wfilename) char *args, *wfilename;   /* open writefile: no error */
{                                               /* file exists */
int file;
        args=skiparg(args);
        args=skipbl(args);
        if (syschkfn(args)==ERR) {
                return(ERR);
        }
        syscopfn(args,wfilename);
        if ((file=sysopen(wfilename,"w"))==ERR) {
                return(ERR);
        }
        writefile = file;
        return(OK);
}
/***** name1() unchanged *****/

name3(args,filename) char *args, *filename;
{
        if (syschkfn(args)==ERR) {
                return(ERR);
        }
        syscopfn(args,filename);
        return(OK);
}
writel(args) char *args;        /* new: write first n lines of buffer to */
{                               /* writefile and delete from buffer */
int n, to;
        if (writefile == -1) {
                message("file not opened");
                return;
        }
        if (getarg(args,&to)==ERR) {
                return;
        }
        if (bufgo(1)==ERR) {
                sysclose(writefile);
                writefile = -1;
                wfilename[0] = EOS;
                fmtwfile(wfilename);
                return;
        }
        while ((bufatbot()==NO)&(bufln()<=to)) {
                n=bufgetln(databuf,MAXLEN);
                n=min(n,MAXLEN);
                if (pushline(writefile,databuf,n)==ERR) {
                        return;
                }
                if (bufdn()==ERR) {
                        return;
                }
        }
        if (bufgo(1)==ERR) {
                return;
        }
        if (bufndel(to)==ERR){
                return;
        }
        bufout(bufln(),1,SCRNL1);
        if (bufatbot()) {
                bufsaved();
        }
        bufgo(1);
}
save()
{
int n, oldline;
        if (writefile == -1) {                  /* changed to writefile */
                message("file not opened");     /* file assumed already open */
                return;
```

Screen Editor Enhancements

(Listing continued, text begins on page 38)

```
        }
        oldline=bufln();
        if (bufso(1)==ERR) {
                sysclose(writefile);    /* close the writefile if error */
                writefile = -1;
                wfilename[0]=EOS;
                pmtwfile(wfilename);
                return;
        }
        while (bufatbot()==NO) {
                n=bufgetln(databuf,MAXLEN);
                n=min(n,MAXLEN);
                if (pushline(writefile,databuf,n)==ERR) {
                        break;
                }
                if (bufdn()==ERR) {
                        break;
                }
        }
        if (bufatbot()) {
                bufsaved();
        }
        bufso(oldline);
                                        /* writefile no longer closed */
}
closeread()                             /* new: close the readfile */
{
        if (readfile!= -1) {
                sysclose(readfile);
                readfile= -1;
                rfilename[0]=EOS;
                pmtrfile(rfilename);
        }
}
closewrite()                            /* new: close the writefile */
{
        if (writefile != -1) {
                sysclose(writefile);
                writefile= -1;
                wfilename[0]=EOS;
                pmtwfile(wfilename);
        }
}

/***** search() unchanged *****/

/***** the only alterations to search1() are in amatch() references
       1st reference was: if(amatch(line,pat+1,0)==YES) {
                  is now: if (amatch(line,pat+1,NO)==0) {
       2nd reference was: if (amatch(line,pat,col++)==YES) {
                  is now: if ((col=amatch(line,pat,YES))>=0) {
       Also - search1() is now called suurch()
*****/

/***** tabs() and chkbuf() are unchanged *****/

getarg(args,val) char *args; int *val;  /* new: get a single argument */
{
        args=skiparg(args);
        args=skipbl(args);
        if (*args==EOS) {
                *val=1;
                return(ERR);
        }
        if (number(args,val)==NO) {
                message("bad argument");
                return(ERR);
        }
        topline=max(1,bufln()-SCRNL2);
        bufout(topline,2,SCRNL2);
        bufso(topline);
}
moveit (args) char *args;       /* new: moves "lenn" lines from "from" */
{                               /* to "to" - same as copyit except deletes */
int from,to,lenn;               /* old lines */
int topline;
        if (get3args(args,&from,&to,&lenn)==ERR) {
                return;
        }
        if (to == from) {
                return;
        }
        if ((to>from)&(to<=(from+lenn))) {
                message("interleaving not permitted");
                return;
        }
        bufcopy(from,to,lenn);
        if (to>from){
                bufso(from);
                bufndel(lenn);
        }
        else {
                bufso(from+lenn);
                bufndel(lenn);
        }
        bufso(1);
        topline=max(1,bufln()-SCRNL2);
        bufout(topline,2,SCRNL2);
        bufso(topline);
}
```

```
/* ED4.C */

#include ed0.c
#include ed1.ccc
char    editbuf[MAXLEN];
int     editp;
int     editpmax;
int     edcflag;

/***** edabt() and edbegin() unchanged *****/

ederase()       /* new: erase from cursor to end of line */
{
        if (editp==editpmax) {
                return;
        }
        edcflag=YES;
        editpmax=editp;
        edredraw();
}
edhome()        /* new: home cursor.  subsequent calls alternate */
{               /* cursor to top and bottom of screen */
int ypos;
        if (edrepl()!=OK){
                return;
        }
        if ((ypos=outyget())<SCRNL3) {
                while((ypos<SCRNL1)&&(!bufnrbot())){
                        if (bufdn()!=OK){
                                return(ERR);
                        }
                        ypos++;
                }
```

```
        }
        return(OK);
}
/* new: get three arguments                                          */
set3args(args,val1,val2,val3) char *args; int *val1, *val2,*val3;

{
        args=skiparg(args);
        args=skipbl(args);
        if (*args==EOS) {
                return(ERR);
        }
        if (number(args,val1)==NO) {
                message("bad argument");
                return(ERR);
        }
        args=skiparg(args);
        args=skipbl(args);
        if(*args==EOS) {
                return(ERR);
        }
        if (number(args,val2)==NO) {
                message("bad argument");
                return(ERR);
        }
        args=skiparg(args);
        args=skipbl(args);
        if(*args==EOS) {
                return(ERR);
        }
        if (number(args,val3)==NO) {
                message("bad argument");
                return(ERR);
        }
        else {
                return(OK);
        }
}
/***** set2args(), skiparg(args), and skipbl() unchanged *****/

chkkey()
{
int c;
        c=syscstat();
        if (c==-1) {            /* note correction from original version */
                return(NO);
        }
        else if (c==' ') {
                pmtline();
                if (syscin()==' ') {
                        return(NO);
                }
        }
        return(YES);
}
/***** amatch() is now machine language code in ED8.c *****/

/***** replace() is unchanged *****/
copyit (args) char *args;       /* new: copies "lenn" lines from "from"   */
{                               /* to "to" */
int from,to,lenn;
int topline;
        if (set3args(args,&from,&to,&lenn)==ERR) {
                return;
        }
        if (to == from) {
                return;
        }
        if ((to>from)&(to<=(from+lenn))) {
                message("interleaving not permitted");
                return;
        }
        bufcopy(from,to,lenn);
        bufgo(1);
        }
        else {
                while((ypos>1)&&(!bufattop())){
                        if (bufup()!=OK){
                                return(ERR);
                        }
                        ypos--;
                }
        }
        edgetln();
        outxy(edxpos(),ypos);
}
edchng(c) char c;
{
char oldc;
int k;
        if (editp>=editpmax) {
                edins(c);
                return;
        }
        oldc=editbuf[editp];
        editbuf[editp]=c;
        fmtadj(editbuf,editp,editpmax);
        k=fmtlen(editbuf,editpmax);
        if (k>SCRNW1) {
                editbuf[editp]=oldc;
                fmtadj(editbuf,editp,editpmax);
        }
        else {
                edcflag=YES;
                editp++;
                if (c==TAB) {      /* only need to redraw if TAB char */
                edredraw();
                }
                else {
                fmtchdev(c);
                }
        }
}
/***** eddel() in original version now called edldel(): no change in code *****/

edcdel()  /* deletes character at cursor - edldel() deletes prior character */
{
int k;
        if (edexpos() < outxset()) {
                outxy(outxset()-1,outyset());
                return;
        }
        edcflag=YES;
        if (editp == editpmax)  {
                if (editp==0) {
                        return;
                }
                editp--;
                editpmax--;
                edredraw();
                return;
        }
        k=editp;
        while (k<(editpmax-1)) {
                editbuf[k]=editbuf[k+1];
                k++;
        }
        editpmax--;
        edredraw();
}

/***** all remaining ED4.C functions unchanged *****/
```

```
/* ED5.C */
```

(Continued on next page)

Screen Editor Enhancements

(Listing continued, text begins on page 38)

```
#include ed0.c
#include ed1.ccc
int fmtrev;
int fmttab;
int fmtdev;
int fmtwidth;
int fmtcol[MAXLEN1];
fmtytab()     /* new - sets mode to show tabs in reverse video */
{
        fmtrev=YES;
}
fmtntab()     /* new - sets normal tab mode (default) */
{
        fmtrev=NO;
}
/***** fmtassn(),fmtadj(), and fmtlen() unchanged *****/
fmtsubs(buf,i,j) char *buf; int i,j;
{
int k;
        if (fmtcol[i]>=fmtwidth) {
                return;
        }
        outxy(fmtcol[i],outyget());
        while (i<j) {
                if (buf[i]==CR) {
                        break;
                }
                if (fmtcol[i+1]>fmtwidth) {
                        break;
                }
                fmtoutch(buf[i],fmtcol[i]);
                i++;
        }
        if (fmtcol[i]<SCRNW1) {    /* modified to allow full screen width */
        outdeol();
        }
}
fmtsout(buf,offset) char *buf; int offset;
{
char c;
int col,k;
        col=0;
        while (c= *buf++) {
                if (c==CR) {
                        break;
                }
                k=fmtchlen(c,col);
                if ((col+k+offset)>fmtwidth) {
                        break;
                }
                fmtoutch(c,col);
                col=col+k;
        }
        return(col);            /* fmtsout() now returns column value */
}
fmtchlen(c,col) char c; int col;
{
        if (c==TAB) {
                return(fmttab-(col%fmttab));
        }
        else if ((c<32)&(fmtdev==NO)) {     /* if tabs not displayed */
                return(2);
        }
        else {
                return(1);
        }
}
fmtoutch(c,col) char c; int col;
{
}
outdnhas()
{
        return(YES);
}
outsup()
{
        outxy(0,SCRNL1);
        sysdelay();             /*to control scroll rate */
        syscout(10);
}
outsdn()
{
        outxy(0,0);
        sysdelay();              /* to control scroll rate */
        syscout(27);             /* H19/H89 */
        syscout('L');
}
```

```
/* ED7.C */

#include ed0.c
#include ed1.ccc
char pmtln[MAXLEN];
char pmtrfn[SYSFNMAX];          /* now both read and write filenames */
char pmtwfn[SYSFNMAX];
pmtmess(s1,s2) char *s1, *s2;
{
int x,y;
        x=outxget();
        y=outyget();
        outxy(0,0);
        outdelln();
        pmt1line();
        pmt1col(x);
        fmtsout(s1,outxget());
        fmtsout(s2,outxget());
        syscin();
        pmt1line();
        pmt1col(x);
        pmt1file(pmtrfn);      /* both read and write files */
        pmt2file(pmtwfn);
        pmt1mod(pmtln);
        outxy(x,y);
}
pmtmode(s) char *s;
{
int x,y;
        x=outxget();
        y=outyget();
        outxy(0,0);
        outdelln();
        pmt1line();
        pmt1col(x);
        pmt1file(pmtrfn);
        pmt2file(pmtwfn);      /* read and write files */
        pmt1mode(s);
        outxy(x,y);
}
pmtrfile(s) char *s;     /*changed: readfile */
{
int x,y;
        x=outxget();
        y=outyget();
```

```
int k;
        if (c==TAB) {
                k=fmtchlen(TAB,col);
                if ((fmtdev==NO)&(fmtrev==YES)) {
/*show tab characters in reverse video */
                        outchar(ESC1);
                        outchar('p');
                }
                while ((k--)>0) {
                        fmtchdev(' ');
                }
                if ((fmtdev==NO)&(fmtrev==YES)) {
                        outchar(ESC1);
                        outchar('q');
                }
        }
        else if (c<32) {
/* show control characters in reverse video */
                if (fmtdev==NO) {
                        outchar(ESC1);
                        outchar('p');
                        fmtchdev(c+64);
                        outchar(ESC1);
                        outchar('q');
                }
                else {
                        fmtchdev('^');
                        fmtchdev(c+64);
                }
        }
        else {
                fmtchdev(c);
        }
}

/***** fmtchdev(), fmtcrlf(), and fmtset() unchanged *****/
-------------------------------------------------------------------------------

/* ED6.C */

#include ed0.c
#include ed1.ccc
int outx,outy;
/***** outsetx() and outsety() changed in name to outxset() and outyset()***/

/***** outchar() unchanged *****/

outxy(x,y) int x,y;
{
        outx=x;
        outy=y;
        syscout(27);            /* H19/H89 values */
        syscout('Y');
        syscout(y+32);
        syscout(x+32);
}
outclr()
{
        syscout(27);            /* H19/H89 */
        syscout('E');
}
/***** outdelln() unchanged *****/

outdeol()
{
        syscout(27);            /* H19/H89 */
        syscout('K');
}
outuphas()
{
        return(YES);
```

(Continued on page 55, column 2)

```
        outxy(0,0);
        outdelln();
        pmt1line();
        pmt1col(x);
        pmt1file(s);
        pmt2file(pmtwfn);  /* add writefile */
        pmt1mode(pmtln);
        outxy(x,y);
}
pmtwfile(s) char *s;    /* new: writefile */
{
int x,y;
        x=outxget();
        y=outyget();
        outxy(0,0);
        outdelln();
        pmt1line();
        pmt1col(x);
        pmt1file(pmtrfn);
        pmt2file(s);
        pmt1mode(pmtln);
        outxy(x,y);
}
pmtedit()
{
        pmtmode("edit:   ");    /* add blanks to assure erasing "command" */
}
pmtline()
{
int x,y;
        x=outxget();
        y=outyget();
        outxy(0,0);
        outdelln();
        pmt1line();
        pmt1col(x);
        pmt1file(pmtrfn);       /* change: separate read and write files */
        pmt2file(pmtwfn);
        pmt1mode(pmtln);
        outxy(x,y);
}

/***** pmtcol() and pmtcmnd() unchanged *****/

pmt1mode(s) char *s;
{
int i;
        outxy(60,0);            /* change of column location */
        fmtsout(s,60);
        i=0;
        while (pmtln[i++]= *s++) {
                ;
        }
}
pmt1file(s) char *s;
{
int i;
        outxy(25,0);
        if (*s==EOS) {
                fmtsout("no rdfile",25);        /* change of message */
        }
        else {
                fmtsout(s,25);
        }
        i=0;
        while (pmtrfn[i++]= *s++) {
                ;
        }
```

(Continued on page 60, column 1)

Screen Editor Enhancements

(Listing continued, text begins on page 38)

```
}
pmt2file(s) char *s;    /* new: write out writefile name or "no wrtfile" */
{
int i;
        outxy(40,0);
        if (*s==EOS) {
                fmtsout("no wrtfile",40);
        }
        else {
                fmtsout(s,40);
        }
        i=0;
        while (pmtwfn[i++]= *s++) {
                ;
        }
}

/***** pmtline1(), pmtcol1() changed in name to pmt1line(), pmt1col() *****/
-------------------------------------------------------------------------------



/* ED8.C */

/* altered for Software Toolworks compiler with M80 option */
#include ed0.c
#include ed1.ccc
cpmcall(o,c) int o; char c;     /* new: cpmcall absent from original listing */
{
#asm
        POP     H       ;RETURN
        POP     D       ;CHAR C -- INPUT OR OUTPUT
        POP     B       ;INT O -- OPCODE
        PUSH    B
        PUSH    D
        PUSH    H       ;REPLACE RETURN
        CALL    5
        MOV     L,A
        MVI     H,0
/*      RET             Note: Software Tookworks compiler supplies RET */
#endasm
}       /* added */
syscstat()
{
int c;
        c=cpmcall(6,-1);
        if (c==0){              /* correction of original version */
                return(-1);
        }
        else {
                return(c);
        }
}
syscin()        /* extensively altered to recognize escape sequences */
{               /* from H19/H89 special keys */
int s,c;
        while ((c=cpmcall(6,-1))==0) {
                ;
        }
        if (c==ESC1) {
                while ((c=cpmcall(6,-1))==0) {
                        ;
                }
                if (c=='?') {
        else if (c==LF) {       /* map LF to CR for Software Toolworks  */
                return(CR);     /* compiler */
        }
        else {
                return(c);
        }
}
syspshch(c,file) char c; int file;
{
        if (c==CR) {            /* map CR to LF for Software Toolworks   */
                if (putc(LF,file)==-1) {        /* compiler */
                        error("disk write failed");
                        return(ERR);
                }
        }
        else if (putc(c,file)==-1) {
                error("disk write failed");
                return(ERR);
        }
        else {
                return(c);
        }
}


/***** syspopch() unchanged *****/


syschkfn(args) char *args;
{
        if (args[0] == EOS) {   /* add check for zero-length file name */
                message("no file name");
                return(ERR);
        }
        else {
                return(OK);
        }
}



/***** syscopfn() unchanged *****/


sysdnmov(n,d,s) int n,d,s;
{
/* extensively altered to allow LDDR for Z80 processors using parity check
   for processor type.  Also checks for zero-length move to avoid moving
   everything */
#asm
        POP     PSW     ;RETURN
        POP     H       ;SOURCE
        POP     D       ;DESTINATION
        POP     B       ;LENGTH
        PUSH    PSW     ;RESTORE RETURN
        MOV     A,C
        ORA     B
        JZ      SYSDN3
        MVI     A,2
        INR     A
        JPE     SYSDN1
        DB      0EDH,0B8H       ;LDDR INSTRUCTION FOR Z80
        JMP     SYSDN3
;
SYSDN1: MOV     A,B
        ORA     C
        JZ      SYSDN3
        MOV     A,M
        STAX    D
        DCX     H
```

```
                        while ((c=cpmcall(6,-1))==0) {
                                ;
                        }
/* "switch" and "case" construct can be altered to a sequence of          */
/* "else if" statements if compiler lacks "switch".                       */
/* Looks slow, but main body is executed only for escape sequences        */
                        switch (c) {
                                case 'a': return(UP1);
                                case 'x': return(UP2);
                                case 'r': return(DOWN2);
                                case 't': return(LEFT1);
                                case 'v': return(RIGHT1);
                                case 'w': return(INS1);
                                case 'M': return(ESC1);
                                case 'y': return(DEL1);
                                case 'p': return(GTOCH);
                                case 's': return(ZAP1);
                                case 'n': return(DTOCH);
                                case 'u': return(HOME);
                                default: continue;
                        }
                }
                else {
                        switch (c) {
                                case 'S': return(DSCROL);
                                case 'W': return(USCROL);
                                case 'U': return(GOTO);
                                case 'V': return(LSTRT);
                                case 'Q': return(ABT1);
                                case 'P': return(EDIT1);
                                case 'T': return(LEND);
                                case 'R': return(ESC1);
                                case 'J': return(ERASE);
                                default: return(c);
                        }
                }
        }
        else {
                return(c);
        }
}


/***** CP/M VERSION 1.4 ROUTINES OMITTED *****/


/***** syscout() and syslout() unchanged *****/


sysend()
{
#asm
        LHLD    6
        DCX     H
/*      RET                 Note: Software Tookworks compiler supplies RET */
#endasm
}       /* added */


/***** sysopen() and sysclose() unchanged *****/


sysrdch(file) int file;
{
int c;
        if ((c=getc(file))==-1) {
                return(EOF);
        }
```

(Continued on page 60, column 2)

```
        DCX     D
        DCX     B
        JMP     SYSDN1
;
SYSDN3: POP     H
        PUSH    H
        PUSH    H
        PUSH    H
        PUSH    H
/*      RET             Note: Software Tookworks compiler supplies RET */
#endasm
}       /* added */
sysupmov (n,d,s) int n,d,s;

/* extensively altered to allow LDIR for Z80 processors using parity check
   for processor type.  Also checks for zero-length move to avoid moving
   everything */

{
#asm
        POP     PSW     ;RETURN
        POP     H       ;SOURCE
        POP     D       ;DESTINATION
        POP     B       ;LENGTH
        PUSH    PSW     ;RESTORE RETURN
        MOV     A,C
        ORA     B
        JZ      SYSUP3
        MVI     A,2
        INR     A
        JPE     SYSUP1
        DB      0EDH,0B0H       ;LDIR INSTRUCTION FOR Z80
        JMP     SYSUP3
;
SYSUP1: MOV     A,B
        ORA     C
        JZ      SYSUP3
        MOV     A,M
        STAX    D
        INX     H
        INX     D
        DCX     B
        JMP     SYSUP1
;
SYSUP3: POP     H
        PUSH    H
        PUSH    H
        PUSH    H
        PUSH    H
/*      RET             Note: Software Tookworks compiler supplies RET */
#endasm
}       /* added */

amatch (a,b,c) int a,b,c;
/* matches pattern at b to line at a
   all all question marks match
   if flag c is NO match is only at beginning of line.
   Returns column of match if match, otherwise -1
*/
{
#asm
        POP     PSW     ;RETURN ADDRESS
        POP     B       ;FLAG
        POP     D       ;PATTERN ADDRESS
        POP     H       ;LINE ADDRESS
        PUSH    H
        PUSH    D
        PUSH    B
```

(Continued on page 62, column 1)

Screen Editor Enhancements

(Listing continued, text begins on page 38)

```
        PUSH    PSW
        SHLD    LINAD
        XCHG
        SHLD    PATAD   ;PATAD IN HL, LINAD IN DE
        MVI     A,0
        ORA     C
        JNZ     MTCH1
        ORA     B
MTCH1:: PUSH    PSW     ;Z FLAG IS 'NO'
        LXI     B,0
        MOV     A,M
        CPI     0
        JZ      MTCH3   ;ZERO LENGTH MATCH STRING - NOMATCH
MTCH5:: CPI     '?'
        JZ      MTCH2   ;SUCCESS SO FAR
        LDAX    D
        CMP     M
        JZ      MTCH2
        CPI     0       ;END OF LINE
        JZ      MTCH3   ;NOMATCH
        POP     PSW
        PUSH    PSW
        JZ      MTCH3
        LHLD    LINAD
        INX     H
        INX     B
        SHLD    LINAD
        XCHG
        LHLD    PATAD
        MOV     A,M
        JMP     MTCH5
LINAD:: DS      2
PATAD:: DS      2
MTCH2:: INX     H
        INX     D
        MOV     A,M
        CPI     0
        JZ      MTCH4   ;TOTAL SUCCESS
        JMP     MTCH5
MTCH3:: LXI     B,-1
MTCH4:: POP     PSW
        MOV     H,B
        MOV     L,C
#endasm
}
sysdelay()
{
/* sets scrolling rate for scroll up or down - modify by
changing number loaded into D register - smaller value for slower scroll */
#asm
        ANA     A
        LXI     D,10
        LXI     H,0
DOMOR:: DAD     D
        JNC     DOMOR
#endasm
}
```

```
/* ED9.C */

#include ed0.c
#include ed1.ccc
```

```
                return;
        }
        if (bufatbot()) {
                outdeol();
        }
        else {
                c=fmtsout(bufp,0);              /* c= added */
                if (c<SCRNW1) {                  /* now allows full screen */
                outdeol();
                }
        }
}

/****** bufext() unchanged *****/


/* bufstrch is bufext modifed to allow entering beyond buffer with warning */
bufstrch(length) int length;
{
        if ((bufpmax+length)>=bufend) {
                bufdnmov(bufp,bufpmax-1,length);
                bufpmax=bufpmax+length;
                message("Caution: main buffer is full");
                return(ERR);
        }
        bufdnmov(bufp,bufpmax-1,length);
        bufpmax=bufpmax+length;
        return(OK);
}


buflength(first,n) int first, n;    /* new: returns length of n lines from  */
{                                   /* first line specified */
char *start;
int i,len,line;
        if (n<=0) {
                len=0;
                return(len);
        }
        if (bufso(first)==ERR) {
                len=0;
                return(len);
        }
        start=bufp;
        i=1;
        line = first;
        while (i++<=n) {
                if (bufso(++line)==ERR) {
                        len=0;
                        return(len);
                }
        }
        len=bufp-start;
        return(len);
}


bufcopy(from,to,n) int from,to,n;       /* new: copies "n" lines from "from"  */
{                                       /*      to "to" */
int length;
char *source,*dest;
        if (from==to){
                return;
        }
        if ((length=buflength(from,n))==0){
                return;
        }
        if (bufso(from)==ERR){
                return;
        }
        source=bufp;
        if (bufso(to)==ERR){
                return;
```

```
/***** ED9.C functions unchanged *****/
-------------------------------------------------------------------------------


/* ED10.C */

#include b:ed0.c
#include b:ed1.ccc
int bufcflag;
char *bufp;
char *bufpmax;
char *bufend;
int bufline;
int bufmaxln;
/* mbuffer is external for Software Toolworks compiler to force it beyond
   code sections */
extern char mbuffer[1];
bufnew()
{
        bufp=bufpmax=mbuffer+1;
/* need more room for Software Toolworks stack and buffers */
        bufend=sysend()-1500;
        bufline=1;
        bufmaxln=0;
        mbuffer[0]=CR;
        bufcflag=NO;
}

/***** bufln(),bufchg(),bufsaved(),buffree(),bufgo(),bufup(), bufdn(),


                bufins() unchanged  *****/

buflins(p,n) char *p; int n;     /* added: a full buffer does not truncate */
{                               /*   line, but issues warning and returns   */
int k, x;                       /*   ERR - allows editing of large files    */
        x = 0;                  /*   in pieces */
        if(bufstrch(n+1)==ERR) {
                x = 1;
        }
        k=0;
        while (k<n) {
                *(bufp+k)= *(p+k);
                k++;
        }
        *(bufp+k)=CR;
        bufmaxln++;
        if ((n==0)&(bufnrbot())) {
                ;
        }
        else {
                bufcflag=YES;
        }
        if (x == 0) {
                return(OK);
        }
        else {
                return(ERR);
        }
}

/***** bufdel(), bufrepl(), bufgetln(), bufdnmov(), bufupmov(),
       bufatbot(), bufnrbot(), bufattop(), bufout() unchanged *****/

buflnout(line) int line;
{
int c;
        if (bufgo(line)==ERR) {
                fmtsout("disk error: line deleted",0);
                outdeol();
        }
        dest=bufp;
        if ((bufpmax+length)>=bufend) {
                message ("insufficient room in buffer for copy or move");
                return;
        }
        if (bufext(length)==ERR){
                return;
        }
        bufgo(1);
        bufmaxln=bufmaxln+n;
        if (from<to) {
                sysupmov(length,dest,source);
        }
        else {
                sysupmov(length,dest,source+length);
        }
}
-------------------------------------------------------------------------------


;MBUFFER.MAC (REQUIRES M80 TO ASSEMBLE) -- MUST BE VERY LAST MODULE LINKED

        ENTRY MBUFFER
MBUFFER::       DS      0
        END
```

End Listing

(Continued on page 62, column 2)

Shifts and Rotations on the Z80

by Ron Goodman

Ron Goodman, 12702 Emelita Street, North Hollywood, CA 91607.

The shift and rotation commands on the Z80 are powerful commands. They allow simple division and multiplication. Along with the BIT command, they provide powerful control over the eight bits in each byte. But for all the brilliant applications, how does one know which specific type of shift or rotation one needs? Since I could not find an explanation of the discrete differences between these commands, I compiled the following with a little research and experimentation. Hopefully it will prove helpful when employing these powerful commands in any future assembly/machine language endeavors. Note that whenever the command is "xxx r," the "r" refers to any single, unpaired Z80 register other than "F" and a couple of other things. Thus, r can be A, B, C, D, E, H, L, (HL), (IX+d), or (IY+d).

(1) RR r – Rotate Right thru Carry. This command moves each bit in the chosen register to the right. The bit in position 0 goes to the carry flag, and the carry flag status is placed into bit position 7. This command is valuable because it allows for examining each individual bit's status. Each time RR r is performed, the next bit is placed in the carry, allowing appropriate action – e.g., "JR C,BITON." An example of this command may be found in the program in Figure 1 at right.

(2) RLC r – Rotate Left Circular. This moves each bit in the specified register to the left. The bit in position 7 of the register is put in position 0. The use of this rotate is probably less common. It can multiply a number by two if the result is less than 255. If the result is greater than 255, the status of bit 8 (256th's place, which is really the ninth bit and so normally not there) will be stored in bit position 0. Bit position 0 will hence equal 1 if the result is greater than 255, and 0 if the result is less than or equal to 255. Figure 2 at right shows a program using RLC. Notice that it uses the "A" register but it could use any of the "r" values.

(3) RL r – Rotate Left thru Carry. This moves each bit to the left, but the bit in position 7 goes to the carry flag and the carry flag status is put into bit 0. This command can be used like RR r, except that this rotates the other direction. If it is used for multiplication and the result is greater than 255, the carry flag will be set.

(4) RRC r – Rotate Right Circular. This moves each bit to the right. The bit in position 0 of the byte is put into position 7. This command works like RLC r except it rotates the other direction and hence it divides.

(5) SLA r – Shift Left Arithmetic. With this command all of the bits are shifted to the left. The status of bit 0 is always 0 after this shift is used, and bit 7 is lost. This is best described in Table 1 on page 66. This command has a somewhat more useful multiply ability for most applications. It multiplies by two and leaves 0 at bit position 0. After multiplying any binary number by two, bit position 0 should equal 0 because the answer is even.

(6) SRA r – Shift Right Arithmetic. All of the bits are shifted to the right. The status of bit 7 remains unchanged, but bit 0 is lost. Although this command can be used for division, the main problem is that it leaves bit 7 on and hence it will give an inaccurate result if the number to be divided is greater than 127. There may be some practical reason for leaving bit 7 on, but I am unaware of it.

(7) SRL r – Shift Right Logical. This command is very similar to SRA r. All of the bits are again shifted to the right, and bit 0 is lost. But with this command, the status of bit 7 is always set to 0 after a shift. This makes this command far more practical for division than SRAr. Unlike SRAr, which will only give an accurate result if the number being divided is less than 128, this will give an accurate answer if the number to be divided is less than or equal to 255 (which will always be the case if it is being stored in an 8-bit register).

Table 1 shows how each of the commands operates. Each one is performed several times on a number to provide easy visualization of the command's results.

DDJ

(Table 1. on page 66)

```
00100  ;THIS PROGRAM WILL PRINT THE BINARY REPRESENTATION
00110  ;OF THE 'A' REGISTER ON A TRS-80 SCREEN AT
00120  ;SCREEN LOCATION 3C00H (15360 DECIMAL).
00130  ;
00140  PRINT   LD      B,8
00150          LD      HL,3C07H
00160  LOOP    RRA
00170          LD      A,48
00180          JR      NC,ZERO
00190          LD      A,49
00200  ZERO    LD      (HL),A
00210          DEC     HL
00220          DJNZ    LOOP
00230          RET
```

Figure 1.

```
00100  ;PROGRAM TO MULTIPLY 'A' BY 2.  PRODUCT MUST BE
00110  ;LESS THAN OR EQUAL TO 128.
00120  ;THE COMMAND RLCA DOES SAME THING AS RLC  A
00130  ;
00140  MULT2   RLC     A
00150          RES     0,A
00160          RET
```

Figure 2.

RR	THE BYTE	CARRY	RRC	THE BYTE	SRA	THE BYTE	SRL	THE BYTE
	11111111	0		00000001		10100000		10100000
	01111111	1		10000000		11010000		01010000
	10111111	1		01000000		11101000		00101000
	11011111	1		00100000		11110100		00010100
	11101111	1		00010000		11111010		00001010
	11110111	1		00001000		11111101		00000101
	11111011	1		00000100		11111110		00000010
	11111101	1		00000010		11111111		00000001
	11111110	1		00000001		11111111		00000000
	11111111	0		10000000		11111111		

RL	THE BYTE	CARRY	RLC	THE BYTE	SLA	THE BYTE
	00000001	0		00000001		11111111
	00000010	0		00000010		11111110
	00000100	0		00000100		11111100
	00001000	0		00001000		11111000
	00010000	0		00010000		11110000
	00100000	0		00100000		11100000
	01000000	0		01000000		11000000
	10000000	0		10000000		10000000
	00000000	1		00000001		00000000
	00000001	0		00000010		00000000

Table 1.

SBC, TSX and TXS

Instructions of the 6800 and 6502

by B.T.G. Tan

The 6800 and 6502 microprocessors have very similar architectures and instruction sets. The common instructions of these two microprocessors have been listed elsewhere,[1,2] and the purpose of this note is to draw attention to three common instructions which have subtle but important differences in their operation on the two MPUs. These are:

SBC	Subtract with carry (on the 6800 the mnemonic is followed by A or B indicating the accumulator concerned)
TSX	Transfer stack pointer to index register
TXS	Transfer index register to stack pointer

SBC

The differences in the carry-bit operation for the subtract instruction of the 1802 and Z80 microprocessors have been pointed out by Merrin.[3] Likewise, the differences in SBC between the 6800 and 6502 are due to the different ways in which the two MPUs utilize the carry bit of their status registers. Though SBC is nominally described as subtract with carry, a better description would be subtract with borrow.

The add with carry instruction, ADC, is also common to the 6800 and 6502, and functions identically on them. The operation can be represented as

$$A + M + C$$

where A is the contents of the accumulator, M the operand, and C the carry bit of the status register. The result is put into the accumulator and the resulting carry bit is stored in C. The resulting carry bit will be 1 only if the result exceeds 255. In the case of the 6502, ADC is the only addition instruction, and if an add without carry is required, the carry bit must be cleared first by preceding ADC with CLC (clear carry bit).

To understand the SBC instruction better, let us assume that the status registers of both MPUs have another status bit called the borrow bit. The operation of the SBC instruction can then be represented as

$$A - M - B$$

where B is the borrow bit. B, as well as M, is subtracted from A, and the result is put into the accumulator. The borrow bit resulting from the operation will be stored in B. The resulting borrow bit will be 1 only if the absolute value of (M + B) is greater than that of A. The result of the operation goes into the accumulator. The function of B is to facilitate multi-byte subtraction when borrow bits from one 8-bit subtraction have to be utilized in the subtraction for the next higher 8-bits.

If the subtraction without borrow is required, the borrow bit must be cleared before SBC. We can invent two pseudo-instructions

SEB	Set borrow bit
CLB	Clear borrow bit

to enable us to manipulate B, analogously to SEC and CLC for C. Hence we must precede SBC with CLB for a subtract without borrow.

In the 6800 and 6502 MPUs, there is no separate borrow bit in their status registers. The carry bit C does double duty as a borrow bit, so that it is better described as a carry/borrow bit. In the 6800, the borrow bit is identical with the carry bit, so that the SBC operation can be represented as

$$A - M - C$$

and the borrow bit generated is directly put into C. The instructions SEB and CLB correspond to SEC and CLC respectively.

In the 6502, the borrow bit is the inverse of the carry bit, so that SBC is

$$A - M - \overline{C}$$

and the borrow bit generated is inverted before being put into C. Hence the SEB instruction corresponds to CLC and CLB to SEC. For a subtraction without borrow, SBC would have to be preceded by SEC.

The difference is thus best understood by considering the borrow bit B as entirely separate from C and then taking $B = C$ for the 6800 and $B = \overline{C}$ for the 6502. The implementation of the borrow bit for the 6502 is different from the 6800 in order that arithmetic operations on the 6502 would give more consistent results for C.

For example, the arithmetic operations 5 - 3 and 5 + (-3), using the two's complement in the second operation, would lead to different values for C in the case of the 6800. These operations would give the same values for C in the case of the 6502, since its borrow bit is the inverse of C. In Table 1 (below), the two operations are listed for each MPU. showing the corresponding values of the accumulator and the carry bit after each instruction. These operations were verified on SWTPC 6800 and Apple II microcomputers. The 6809, which is the upgraded version of the 6800, appears to implement the borrow bit in a similar manner to the 6800, that is with the borrow bit the same as the carry bit.

TSX and TXS

The transfer instructions between the stack pointer SP and index register X, TSX and TXS are common to both 6800 and 6502 but have important differences. In both MPUs, the current value of SP points to the location one address below the bottom of the stack. These registers are 16-bits long in the 6800, and 8-bits long in the 6502.

In the 6800, TSX increments the contents of SP by 1 and then transfers them to X. The reason for this appears to be to make X point to the actual bottom

6800				6502			
	ACCA	C	B=C		ACC	C	$B=\overline{C}$
LDAA #$05	05	D	D	LDA #$05	05	D	D
CLC	05	0	0	SEC	05	1	0
SBCA #$03	02	0	0	SBC #$03	02	1	0
LDAA #$05	05	D	D	LDA #$05	05	D	D
CLC	05	0	0	CLC	05	0	1
ADCA #$FD	02	1	1	ADC #$FD	02	1	0

Table 1.
5 minus 3 subtraction operations for 6800 and 6502. D indicates don't care state.

B. T. G. Tan, Department of Physics, National University of Singapore.

of the stack. This is presumably for cases when indexing operations on the stack are to be carried out. Conversely, TXS decrements the contents of X by 1 before transferring them to SP, so that if X pointed to the bottom of the stack, SP would then point to the next location below the bottom.

The 6502's TSX and TXS instructions do not perform the increment and decrement operations; they are just straightforward data transfer instructions. Why is this so? SP is an 8-bit register, and its contents ZZ are always interpreted as pointing to the page 1 of memory, $01ZZ. The effective address in indexed addressing is found by adding the 8-bit contents of X to the operand. In the zero-page indexed mode, the effective address is in page zero; in the absolute indexed mode, the page of the effective address depends on the 16-bit operand.

Because of these complications, the TSX and TXS instructions of the 6502 do not appear to be intended to facilitate indexed operations on the stack, as the transfers are effected without incrementing and decrementing. The primary function of TSX and TXS appears to be to effect data transfer to and from SP via X. This is in fact the only way to do so, since the 6502 lacks LDS (load to stack pointer) and STS (store from stack pointer) instructions. These instructions are part of the 6800's instruction set.

The 6809 MPU, which is the upgraded version of the 6800, has the instructions TFR S,X and TFR X,S which are equivalent to TSX and TXS, respectively. In the 6809, the stack pointer points to the bottom of the stack, and not to the location below the bottom as in the case of the 6800. Thus, the two transfer instructions do not perform an increment or decrement before the transfer, and are more like their 6502 equivalents, but for a quite different reason.

Program FLIP

```
comment $

        PROGRAM FLIP

        This program flips from the color-adapter to the monochrome-adapter
        and back again.  The program has no stack segment so that it can be
        converted into a COM file.  This generates an error upon linking which
        should be ignored.

        Simson L. Garfinkel, 1983

        $

data    segment at 40h
        org 16
equip   db ?
data    ends

cseg    segment para 'code'
        org  100h                   ;I want to use this as a COM file
start   proc far
        assume cs:cseg,ss:cseg,ds:data

        mov  ax,cs
        mov  ss,ax
        mov  sp,stacke              ;get end of stack

        push ds
        xor  ax,ax
        push ax                     ;save how to get home
        mov  ax,data                ;use ds to reference segement at 0
        mov  ds,ax
        mov  ah,equip
        and  ah,30h
        cmp  ah,30h                 ;check to see if we are running mono or color
        jne  mono
        jmp  color                  ;we are in mono - switch to color
mono:   or   equip,30h              ;switch to monochrome interface
        mov  ax,2
        int  10h                    ;generate interrupt to reset board
        mov  ah,1
        mov  cx,12*256+13           ;set cursor for lines 12 + 13
        int  10h                    ;
        ret                         ;go back to ms-dos
color:  and  equip,0cfh
        or   equip,010h             ;switch to color interface
        mov  ax,4                   ;clear graphics memory
        int  10h
        mov  al,0
        int  10h                    ;select 40x25 BW text, no color burst
        mov  ah,1
        mov  cx,6*256+7             ;set cursor for lines 6 & 7
        int  10h
        ret
start   endp
        db 16 dup (?)               ;stack area
stacke  label near                  ;End of stack
cseg    ends
        end start
```

Conclusions

The 6800 and 6502 MPUs are so similar that it is worth looking at both together when studying one of them. I have taken this approach in my microprocessor teaching, using a MPU model consisting of the common features of the two MPUs.[2] This MPU model has one 8-bit accumulator, one 8-bit index register, one 16-bit program counter, one 8-bit stack pointer, and one 8-bit status register. It has the same six addressing modes of the 6800, except that indexed addressing is restricted to the zero page only. The instruction set is made up of the 43 instructions common to both 6800 and 6502. These instructions include SBC, TSX, and TXS, and it is important that those using both the 6800 and 6502 understand the small but important differences in these instructions.

References

[1] Levanthal, Lance R. *6502 Assembly Language Programming*, Osborne/McGraw-Hill, 1979, p. 3-107.

[2] Tan, B.T.G. "Common instructions of the 6800 and 6502," *Dr. Dobb's Journal of Computer Calisthenics & Orthodontia*, Volume 4, Number 9, October 1979, pp. 38-39.

[3] Merrin, Stephen. "Addition and Subtraction: The 1802 Versus the Z80," *Byte*, Volume 6, Number 3, March 1981, pp. 224-228. DDJ

SOFTWARE REVIEW

ZAS Z8000
Software Development Package
By Western Wares, P.O. Box C, Norwood, CO 81423
CP/M Version $395
ISIS Version $495
Reviewed by Terry R. Dettmann

What is there to say about an assembler? It either assembles good code or it doesn't. ZAS assembles functioning Z8000 code (consistent with your own ability to design such code in the first place!).

Is there any more to say? Should a review really continue after this point? Well in some cases I'd say no it shouldn't, but in this case there is more to the package that is important to know about. More that the package can do. What more should I say? Well, let's look and see just what you get when you get the assembler package.

First there's the ZAS assembler itself. It is a "full-featured, relocatable cross assembler for the Z8000." Its instruction mnemonics are compatible with those developed by Zilog. Comparing it to Matosian's book on the Z8000 (SYBEX), it didn't always agree on how some things should be done, but it was easy to learn.

ZAS also includes the "extended" instructions needed for some Z8000 units. It wasn't hard to use. It generated good code for the Z8000, but has only really been tested on limited development systems. I haven't tried to code a large project.

ZAS is a two-pass assembler. It has all the standard things you expect to find in a full-featured assembler such as conditional assembly, named sections, segmented and non-segmented addressing support, nested include files, up-to-64-character symbol names, and standard directives and pseudo-ops.

After the assembler comes the ZLK task builder. ZLK is a flexible linking program that allows complicated processes to be set up as overlays and so forth, while allowing the programmer complete control over memory placement of the code modules.

ZLD is an absolute file loader program. Its basic purpose is to load a Z8000 module under CP/M or ISIS so that it can be executed with the ZEX run-time monitor. ZEX allows the user's system to provide run-time support as needed for a Z8000 task. ZEX allows bus-switching between 8080-type CPUs and Z8000 CPUs, as on the IEEE 696 (S-100) bus or the Intel Multibus. It even allows the program to make CP/M or ISIS calls as needed.

In this kind of mode, the Z8000 CPU is acting as if it were a temporary bus master device and the 8080 CPU is the bus master. All I/O is handled by the bus master (the 8080) through the CP/M system.

ZEX is configured by the user to the particular environment. Detailed instructions are provided to allow proper configuration by anyone with reasonable technical knowledge. This isn't a system for the novice Z8000 programmer, but it is a useful tool for serious system development.

Editor's note: Just before press time, we were informed that version 2.0 of the ZAS Z-8000 package had been released. Some of the enhancements will be of particular interest to those familiar with older versions of ZAS. The following new features were added in the new version, which was received too late for review in this issue: new macro pseudo-operations (MACRO, IRP, IRPC, EXITM, REPT, ENDM, LOCAL), other new pseudo-ops (DZ – define zero, DA – define address), new directives ($BASE – set default number base; $LIST, MACON, MACOFF – control macro expansion listing), new comparison opertors taking string or numerical arguments, extended instruction enhancement, more symbol table space and faster searching of tables, more flexibility for input, command-line switches specified by - or /, and ZAS and ZLK now use the same command-line syntax.

6809 Cross Assembler
Available from Avocet Systems Inc., 804 South State Street, Dover, DE 19901
$200
Reviewed by Terry R. Dettmann

It wasn't long ago that I came up with an acute need for a 6809 assembler. Not just a small one either. Further, I was lazy and I wanted full-screen editing ability, flexible printing, and large capacity. In short, I wanted a large system! In fact, all I had was a small system.

It was then that the Avocet ad in some of the computer magazines caught my eye. As a CP/M cross assembler for the 6809, my dreams were fulfilled. I could have all the powerful CP/M tools I wanted and still be able to put things together for a 6809. I could document my code in as much detail as I wanted. I could modularize as much as I wanted. I could work with 6809 code in a convenient way.

It wasn't long before I started putting together some 6809 software including some communications packages and a simple Forth system. I found out just how lazy I had become when I tried to go back to the little 6809 assembler I had. It was good, but just wasn't made for the kind of programming I was doing.

Avocet produces a whole line of cross assemblers which include assemblers for the following processors: 6805, 6809, RCA1802 (COSMAC), Intel MCS-48 & UPI-41 families, Intel 8051, MOS 6502 & 650X family, Motorola 6800/6801 family, NEC 7500 family, Fairchild F8, MOSTEK F8/3870 family, National COP400, and the Zilog Z8 family. Quite a list.

No assembler is worth its salt, though, unless it can produce good code. The Avocet 6809 assembler does as far as I have gone with it.

So far, I haven't run across any errors not traceable to my own programming errors. If the assembler would only assemble what I want instead of what I type. But the point is that it does do what I type!

The basic assember syntax is roughly the same as for a standard 8080 assembler including standard pseudo-ops such as DW, DB, etc. The write-up with the system says that all the assemblers stay with these conventions. I'm not sure I agree that this is the best though I'm sure it makes development and maintenance easier for the Avocet people.

Personally, I would rather have seen the pseudo-ops closer to those used by the standard system for the 6809, but I can live with anything as long as it produces the code I want.

The assembler produces code in one of two output formats, either Intel Hex format or MIKBUG format. I use the Intel Hex format mostly out of habit.

This is a nice assembler with reasonable documentation for the experienced programmer. It carries out the job I want it to do; after all, isn't that what it should do?

DDJ

BOOK REVIEWS

BCPL – The Language and Its Compiler
by Martin Richards
and Colin Whitby-Strevens
Cambridge University Press
173 pages, $10.95
Reviewed by George W. Jolly

They say you can't tell a book by its cover, but sometimes I'm not so sure. The cover of *BCPL* is a gentle interplay of two colors, blue and green, producing results you might think would require three or more inks. *BCPL* is a language very much in that style, favoring careful design over expensive implementation.

BCPL (Basic CPL) is a systems programming language designed by Martin Richards in 1967. The language, developed mostly over the subsequent five years, has been installed on at least twenty-five different computer systems. It has been used for applications including operating systems, compilers, editors, databases – in other words, for a broad spectrum of systems work.

In *The C Programming Language*, Kernighan and Ritchie credit the language BCPL for many of C's most important ideas. BCPL influenced the language B written for UNIX by Ken Thompson in 1970, and B passed this influence on to C. However, BCPL and C are quite different, perhaps most importantly in the handling of data types.

Many modern languages have a variety of data types built into the compiler. Such types might be integer, floating point, character, pointer, and so on. BCPL, on the other hand, has only one type, the machine word. The compiler wastes no time identifying the legality of a construct for the data items involved, because all data items are alike. In BCPL, address variables, or pointers, are easy to use because indirect addressing is a fundamental operator. Any data item may contain an address.

One interesting result of this inexpensive philosophy is that the elements of an array need not all be the same "type." (Try having integers, characters and pointers in the same array in standard FORTRAN!) Other languages do provide this, but as an extra feature. In BCPL it is a consequence of the underlying design. (Funny how FORTH comes to mind at this point.)

Example BCPL routines illustrate the power of the language. One example given is a formatted output procedure accepting a format string (analogous to a FORTRAN format) as the first argument, and the data list as subsequent arguments. A procedure receives its arguments as an array of addresses. It is a simple matter to scan this array with a pointer variable, accessing each parameter in turn. Another pointer variable holds the address of an appropriate editing procedure as indicated by the format string. This is a simple procedure with elegant results.

Other example programs include an interactive debugging program and the lexical/syntax scanner from the BCPL compiler. The latter is discussed in detail, with a page of program facing a page of explanatory text. The applicative expression tree produced by the scanner is an interesting topic in itself.

Another interesting topic of this book is language portability. The authors illustrate a method of bootstrapping the compiler onto a new system by generating a simplified pseudo-code on an existing installation. They also discuss common library procedures that exist in most BCPL installations. These topics may be of interest to implementors of other languages.

In summary, *BCPL* is a well-written book serving several possible needs. It describes an influential (and useful) computer language, illustrates an elegant programming philosophy, and gives practical examples of compiler techniques. The authors' straightforward explanations, like the cover design, complement the elegant simpilicty of BCPL.

User's Guidebook to Digital CMOS Integrated Circuits
by Eugene R. Hnatek
McGraw-Hill
339 pages, $29.00
Reviewed by Eunice B. Stetson

User's Guide to Digital CMOS Integrated Circuits by Eugene R. Hnatek is not a book for the uninitiated. It is most likely to appeal to design engineers and sophisticated hobbyists. It assumes some knowledge of semiconductors, small- and medium-scale logic circuits, conversions between digital and analog signals, and computer hardware. Very little knowledge of mathematics beyond the ability to read complex graphs is assumed. The book is very practical in orientation. Many applications, with schematics, manufacturers' names, and part numbers, are included.

The author makes clear the electrical characteristics of CMOS as compared with other IC logic families. He does not pretend to discuss all CMOS integrated circuits but concentrates instead on medium- and large-scale IC's developed since 1977.

The author uses many schematics and graphs and some charts and timing diagrams to explain the exciting and promising developments in CMOS and SMOS/SOS (silicon on sapphire) technology and applications in telecommunications, computers and process controllers. He envisions many new uses for CMOS large-scale integration, particularly in the interfacing of analog and digital signals on a single IC. Currently, applications of CMOS microprocessors are concentrated in the automotive, games, and TV industries rather than in the computer field.

The author does an outstanding job of making his graphs understandable with labelling and notes on the graphs. Circuit diagrams for MSI and LSI integrated circuits are arranged in a logical sequence with explanation.

In discussing flip-flops, memory cells, and shift registers, the author uses schematics with FET symbols rather than logic symbols. Later he treats counters, timers, multiplexers, comparators, decoders, VARTs, and microprocessors as black boxes. As a teacher I would have preferred a more orderly progression from FETs to logic symbols to flip-flops to shift registers, multiplexers, etc. as the basic building blocks of which circuits are composed. However, in that this book is intended as a reference book and not a text book, this should not be a problem. As with all books which use manufacturers' data and diagrams, there are differences in symbols and conventions with which the reader is used to coping.

Generally, the author's explanations are clear, though as with most authors what follows "it can be seen that . . ." is not always obvious. The author clearly knows his subject well. At times he makes beautifully clarifying statements. This book, compared with others in its field, is clearly written, supplemented with many excellent pictorials, and does an authoritative job in the areas covered.

(Continued on page 72)

CP/M EXCHANGE

by Robert Blum

Digital Research created a stir when they announced CP/M Plus (alias CP/M 3) several months ago. Preliminary speculation on the system's content ranged from a simple maintenance release to full concurrent operation.

If you consider yourself a hacker, a unique individual who treats inadequacy as an opportunity, CP/M Plus will be bittersweet. Many of its performance options are and have been available in the public domain for years. If nothing else, CP/M Plus verifies that when hackers speak, the manufacturers listen. It just takes them a little while to react.

Even though I had hoped for more, the official release is anything but disappointing. It is chock-full of new features geared to performance and versatility, and is a major departure from the CP/M we have grown to love and respect. DR views CP/M Plus as a new product and it will be maintained and marketed separately. At least for now, it will not obsolete CP/M 2.2.

Over the next few months I will be covering various aspects of CP/M Plus: does it work as advertised; how difficult is it to implement; is it worth a $350 investment? Getting to know a new product cannot be effectively accomplished alone. Share your experiences and thoughts with the rest of us by dropping me a note or calling me in the evening. Your input will be appreciated by one and all.

Before getting into this month's topic, I need some help. Is there a patch for CP/M 2.2's CCP that will automatically search user area 0 for a .COM file that was not found in the current user area? If you have one, please send it along or point me to where I can find it.

After a long wait, CP/M Plus is now being shipped. By the time this column is published, it will probably be available through mail order and possibly from a few manufacturers. This estimate may be a little optimistic because implementation of all CP/M Plus's new features requires considerable attention to both hardware and software.

Even though I have not received my own distribution version of CP/M Plus, I have been able to gather up two of the three manuals and spend a few hours using it. What a pleasant surprise! The documentation has been completely rewritten and downsized to wedgy format. Each topic is thoroughly discussed in plain English, with only minimal use of buzz words, and is accompanied by plenty of examples.

Sitting at the keyboard for the first time was disappointing. I had hoped for concurrent operation, or at least a built-in print spooler. CP/M Plus remains single user and still prompts with the familiar A>. I soon learned, however, that all other aspects of the system are new, and verified that this is a completely new product. After only a few minutes at the keyboard I was comfortable with the operator interface and swear by the additional line-editing features. For those who are new to CP/M, the HELP command, which displays summarized information on all the commands, will undoubtedly prove invaluable.

The actual performance of the system, and which features can be implemented, is predicated on how the system is generated. As a replacement for CP/M 2.2 in a standard 64K or less environment (nonbanked), CP/M Plus offers less than 96K of banked memory when available.

Command-line editing on nonbanked systems remains the same as CP/M 2.2, but is greatly improved on banked systems. In the banked environment it is no longer necessary to retype a command if a mistake is made. Simply move the cursor to the error and correct it by inserting or deleting characters. Even after hitting return, commands can be redisplayed and edited provided a transient program was not executed. Refer to Table 1 on page 76 for a complete comparison of the available editing functions.

A host of new commands has been added to those that are carried over from older versions of CP/M. DIRectory, ERAse, REName, TYPe, and USEr are improved while DIRSys has been added to display system files which previously required the use of STAT. Depending on command-line options, transient programs may be automatically loaded to assist the built-in commands of the CCP. I have summarized the built-in commands in Table 2 (page 76). The descriptions used for each command are excerpts from DR's manuals. I do this so that you can sample the flavor of the new documentation.

Enhanced command editing is only a part of the beauty of CP/M Plus. Many new utilities have been added and those that were retained from older versions are now more useful. Also included is the entire RMAC program development package which previously cost $200. Refer to Table 3 (page 78) for a summary of the utility programs.

Before generating the system, you are faced with a dilemma. The minimum system configuration has been increased to 32K and CP/M Plus requires 8.5K. By the time you add in the size of your BIOS a great deal of memory has been taken up, even when a full 64K is available. To make full use of the system requires at least 96K of bank-select memory. Through the use of this additional memory, it is entirely possible to have a 62K TPA because only a minimum BIOS and BDOS are resident in the application bank. The real controlling logic is placed in the other bank along with directory hash tables and sector buffers. Because of this, I would anticipate the introduction of more 8-bit machines with 256K of memory in the not-too-distant future.

Disk space reserved for the system tracks has been reduced by placing the operating system components into regular disk files. This will be especially important for those systems that require very large BIOSs. When booting the system, a small loader from the system tracks is read into memory, which then completes the loading process. In light of more common use of hard disks, individual file size has been increased to 33 megabytes and disk capacities of 512 megabytes are possible.

Generation of the system is handled by a transient program called GENCPM which combines the various system components with your BIOS. The generation process is remarkably similar to that used with MP/M, as are most other features of CP/M Plus.

CP/M Plus is not for everyone. Its initial cost is high and it requires expanded hardware resources if it is to be used to its fullest extent. But the costs associated with its implementation will not seem nearly as high if you need all the development tools which are included. Functionalism is the keyword to describing this new offering from Digital Research.

Next month I will continue with the internals of CP/M Plus and any experiences that I hear from you. You can reach me by phone at (404) 449-8948. DDJ

(Tables I and II on page 76)
(Table III on page 78)

Portions copyright © 1982 by Digital Research; used with permission.

TABLE I

Nonbanked Command Editing

Ctrl-E Forces a physical carriage return but does not send the command line to CP/M 3. Moves the cursor to the beginning of the next line without erasing your previous input.

Ctrl-H Deletes a character and moves the cursor left one character position.

Ctrl-I Moves the cursor to the next tab stop. Tab stops are automatically set at each eighth column. Has the same effect as pressing the tab key.

Ctrl-J Sends the command line to CP/M 3 and returns the cursor to the left of the current line. Has the same effect as a RETURN or a Ctrl-M.

Ctrl-M Sends the command line to CP/M 3 and returns the cursor to the left of the current line. Has the same effect as a RETURN or a Ctrl-J.

Ctrl-R Places a # at the current cursor location, moves the cursor to the next line, and displays any partial command you typed so far.

Ctrl-U Discards all characters in the command line, places a # at the current cursor position, and moves the cursor to the next command line.

Ctrl-X Discards all the characters in the command line, and moves the cursor to the beginning of the current line.

Banked Command Editing

Ctrl-A Moves the cursor one character to the left.

Ctrl-B Moves the cursor to the beginning of the command line without having any effect on the contents of the line. If the cursor is at the beginning, Ctrl-B moves it to the end of the line.

Ctrl-E Forces a physical carriage return but does not send the command line to CP/M 3. Moves the cursor to the beginning of the next line without erasing the previous input.

Ctrl-F Moves the cursor one character to the right.

Ctrl-G Deletes the character indicated by the cursor. The cursor does not move.

Ctrl-H Deletes a character and moves the cursor left one character position.

Ctrl-I Moves the cursor to the next tab stop. Tab stops are automatically set at each eighth column. Has the same effect as pressing the tab key.

Ctrl-J Sends the command line to CP/M 3 and returns the cursor to the beginning of a new line. Has the same effect as a RETURN or a Ctrl-M keystroke.

Ctrl-K Deletes to the end of the line from the cursor.

Ctrl-M Sends the command line to CP/M 3 and returns the cursor to the beginning of a new line. Has the same effect as a RETURN or a Ctrl-J keystroke.

Ctrl-R Retypes the command line. Places a # at the current cursor location, moves the cursor to the next line, and retypes any partial command you typed so far.

Ctrl-U Discards all the characters in the command line, places a # at the current cursor position, and moves the cursor to the next command line. However, you can use a Ctrl-W to recall any characters that were to the left of the cursor when you pressed Ctrl-U.

Ctrl-W Recalls and displays previously entered command line both at the operating system level and within executing programs, if Ctrl-W is the first character entered after the prompt. Ctrl-J, Ctrl-M, Ctrl-U, and RETURN define the command line you can recall. If the command line contains characters, Ctrl-W moves the cursor to the end of the command line. If you press RETURN, CP/M 3 executes the recalled command.

Ctrl-X Discards all the characters left of the cursor and moves the cursor to the beginning of the current line. Ctrl-X saves any characters right of the cursor.

TABLE II

Built-in Commands

DIR The DIR command displays the names of files and the attributes associated with the files. DIR and DIRSYS are built-in utilities; DIR with options is a transient utility.

ERASE The ERASE command removes one or more files from a disk's directory in the current user number. Wildcard characters are accepted in the filespec. Directory and data space are automatically reclaimed for later use by another file. The ERASE command can be abbreviated to ERA.

RENAME The RENAME command lets you change the name of a file that is cataloged in the directory of a disk. It also lets you change several filenames if you use wildcards in the filespecs. You can abbreviate RENAME to REN.

TYPE The TYPE command displays the contents of an ASCII character file on your screen. The PAGE option displays the console listing in paginal mode, which means that the console listing stops automatically after listing N lines of text, where N is usually the system default of 24 lines per page.

USER The USER command sets the current user number. When you start CP/M 3, 0 is the current user number. You can use a USER command to change the current user number to another in the range 0-15.

(Table III begins on page 78)

Portions copyright © 1982 by Digital Research; used with permission.

TABLE III

Utilities

COPYSYS The COPYSYS command copies the CP/M 3 system from a CP/M 3 system disk to another disk with the same format as the original system disk. For example, if the system disk is single density, the disk you copy onto must also be in single-density format.

DATE The DATE command is a transient utility that lets you display and set the date and time of day. When you start CP/M 3, the date and time are set to the creation date of your CP/M 3 system. Use DATE to change this initial value to the current date and time.

DEVICE The DEVICE command is a transient utility that displays current assignments of logical devices and the names of physical devices. This command also sets the communications protocol and speed of a peripheral device.

DUMP DUMP displays the contents of a file in hexadecimal and ASCII format.

ED The ED utility is a line-oriented context editor. This means that you create and change character files line-by-line, or by referencing individual characters within a line.

GENCOM The GENCOM command is a transient utility that creates a special COM file with attached RSX files. RSX files are used as Resident System eXtensions. GENCOM places a special header at the beginning of the output program file to indicate to the system that RSX loading is required. It can also set a flag to keep the program loader active.

GET The GET command is a transient utility that directs CP/M 3 to take console input from a file. The file can contain CP/M 3 system commands and/or input for a user program. If you use the SYSTEM option, GET immediately takes the next system command from the file.

HELP The HELP command is a transient utility that provides information for all of the CP/M 3 commands described in the manual. In the distributed CP/M 3 system, HELP presents *general* information on a command as a topic, and *detailed* information as a subtopic. HELP with no command tail displays a list of all the available topics. HELP with a topic in the command tail displays information about that topic, followed by any available subtopics. HELP with a topic and a subtopic displays information about the specific subtopic.

HEXCOM The HEXCOM command is a transient utility that generates a command file (filetype COM) from a hex input file. It names the output file with the same filename as the input file but with filetype COM. HEXCOM always looks for a file with filetype HEX.

INITDIR The INITDIR command can initialize a disk directory to allow date and time stamping of files on that disk or remove date and time stamps.

LIB The LIB command is used to create and maintain a library of relocatable object modules. Use the LIB utility to create libraries and to append, replace, select, or delete modules from an existing library. You can also use LIB to obtain information about the contents of library files.

LINK The LINK command combines relocatable object modules such as those produced from RMAC into a .COM file ready for execution. Relocatable files can contain external references and publics. Relocatable files can reference modules in library files. LINK searches the library files and includes the referenced modules in the output file.

MAC The MAC utility assembles .ASM disk files into .HEX object files. It also has the ability to process macros.

PATCH The PATCH command displays or installs patch N to the CP/M 3 system or command files.

PIP PIP is a transient utility that copies one or more files from one disk and/or user number to another. PIP can rename a file after copying it; combine two or more files into one file; and copy a character file from one disk to the printer or other auxiliary logical output device. PIP can create a file on disk of input from the console or other logical input device. PIP can also transfer data from a logical input device to a logical output device, thus the name Peripheral Interchange Program.

PUT The PUT command is a transient utility that lets you direct console output or printer output to a file. PUT allows you to direct the system to put console output or printer output to a file for the next system command or user program entered at the console. Or PUT directs all subsequent console or printer output to a file when you include the SYSTEM option.

RMAC The RMAC utility is operationally comparable to MAC except its output is in .REL form.

SAVE The SAVE command copies the contents of memory to a disk file. To use the SAVE utility, first issue the SAVE command, then run your program which reads a file into memory. When your program exits, it exits to the SAVE utility. The SAVE utility prompts you for the filespec to which the memory is to be copied, and the beginning and ending address of the memory to be saved.

SET The SET command initiates password protection and time stamping of files in the CP/M 3 system. It also sets file and device attributes, such as the Read-Only, SYS, and user-definable attributes. It lets you label a disk and password-protect the label.

SETDEF The SETDEF command lets you display or define the disk search order, the temporary drive, and the filetype search order.

SHOW The SHOW utility displays the disk drive attributes.

SID The SID utility replaces the DDT program. It offers more commands and flexibility.

SUBMIT The SUBMIT command lets you execute a group or batch of commands from a SUB file, which is a file with filetype of SUB.

TYPE The TYPE utility displays the contents of an ASCII disk file on your screen. A paging option is now available.

XREF The XREF utility provides a cross-reference summary of variable usage in an assembler program.

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

MS-DOS vs. CP/M-86

Jim Howell of San Jose, California, has written me a couple of interesting letters regarding my coverage of the two major operating systems for the 8086/88 microcomputer family. I have taken the liberty of editing some of his comments and my replies into the form of a dialogue for this column, and hope he will forgive me for the slight rearrangements and changes of wording that were necessary.

JH: In the December issue of *DDJ*, you say that the reason for "excessive" coverage of the 8086/88 is a table that indicates more 8086/88s in use than other 16-bit processors. First, the numbers in that table do not necessarily indicate the level of interest in the various processors. For example, I am sure there are many, including myself, who do not have a 68000-based computer, but are nevertheless interested in that processor. In any case, if you are going to use the numbers in that table to justify the contents of your column, you should be devoting 78% of your column to the 8086/88, 15% to the 68000, and 7% to the Z8000. (What about the TI-9900 and National's 16000?).

Second, if you are going to devote 100% of your column to the 8086/88 (which is the case so far), perhaps the column should be called "8086/88 Toolbox" or even "IBM PC Toolbox." The title "16-Bit Software Toolbox" promises broader coverage than just a single processor. Actually, I suspect that the real reason for covering only the 8086/88 is that you own (or have a lot of access to) an IBM PC and have little or no access to systems that use other 16-bit processors.

RD: You certainly have penetrated right to the heart of the matter, since it is true that until very recently I didn't have any access to a 68000 system. But again, this is relevant to the content of the column and is not a coincidence. The long-range objective of the "16-Bit Software Toolbox" is to provide useful subroutines and utilities for the most commonly used 16-*bit* microprocessors. Although Motorola's 68000 architecture and instruction set are elegant, their decision to make a total break with the architecture of their 8-bit microprocessor family has caused a tremendous lag in the appearance of affordable operating systems and software development tools for 68000-based personal computers.

I do plan to provide increasing coverage of the 68000 in the coming months, and now have both a Cromemco Dual Z80/68000 CPU and a Godbout 68000 CPU to work with. But I'm still waiting for my copy of CP/M-68K, which was announced and demonstrated at the CP/M-83 show in January but is still not being shipped to customers.

JH: I have noticed that your recent columns have criticized various aspects of MS-DOS (apparently you are "pushing" CP/M-86, for some reason). We have used an IBM PC at work for the past several months, using only PC-DOS, and have had no problems with it.

RD: I do not have any vested interest in either CP/M-86 or MS-DOS, and am not trying to "push" one operating system in preference to another. Since MS-DOS presently dominates the 8086/88 user base, I do feel that its problems (and ways of dealing with them) should receive more space in the column. It is true that I feel the Digital Research family of operating systems and language compilers is much more "robust" than the Microsoft products; this is only an opinion, but it is based on many published comparisons and compiler benchmarks, and on my own extensive experience implementing systems tools on CP/M, CP/M-86, and MS-DOS.

JH: Benchmarks published in the November 1982 column showed PC-DOS outperforming CP/M-86 on two out of three disk-intensive tests. This was followed immediately by several paragraphs entitled "And Now the Bad News" about how changing disks in the middle of an operation under PC-DOS could clobber data on a disk, but with CP/M-86 all you get is a harmless little error message. You neglected to state that any editing you may have done to the file under CP/M-86 is lost. Perhaps that seemed too obvious to mention, but it would have balanced the presentation a little.

In January 1983, a letter from a reader appeared on the above subject and on how to recover most of the data on the damaged disk. This section had the rather negative title: "Coping with PC-DOS." The most eye-catching line in Isaac Davidian's letter was: "Unfortunately, I had no backup. . . ." If this reader had had a backup, he would have had much less of a problem. I have not used CP/M-86, but I used CP/M-80 at a previous job and found its READ-ONLY DISK error rather annoying. I always wondered why the system couldn't automatically do whatever was required to make the disk not read-only, instead of making the user start over. CP/M-86 apparently handles new disks as READ-ONLY also.

RD: Mr. Howell correctly points out that, if you switch disks at an inappropriate time under CP/M-86, you will lose the contents of the file you are currently working on. However, this is all you will lose. The integrity of the other files on both the new and previous disks is carefully protected by the operating system. The inexcusable part of PC-DOS's handling of new disks is not that *you can lose* your currently accessed file, but you can lose *all* of the files on the new disk; and the directory is trashed in such a manner that it is nearly impossible to reconstruct it.

This is in direct contradiction to the PC-DOS manual, page D-12, under Function 10H Close File, which states: "This function must be called after file writes to insure [sic] all directory information is updated. On entry, DS:DX point to an opened FCB. The disk directory is searched and if the file is found, its position is compared with that kept in the FCB. If the file is not found in its correct position in the directory, it is assumed the diskette was changed and [register] AL returns X'FF'. Otherwise, the directory is updated to reflect the status in the FCB and AL returns 00." If PC-DOS *really* handled file closure the way the manual states, everything would be peachy.

As for Mr. Davidian's lack of backups, we are all human and I am sure we have all failed at one time or another to make backup disks as often as would be desirable. The operating system ought to help protect us from our frailties, not exploit them!

JH: My own feelings on CP/M-86 vs. PC-DOS are based on Dave Cortesi's column of several months ago, on my four years of working with CP/M-80, and on six months of working with PC-DOS. CP/M-86 is, as I understand, not much more than a direct copy of CP/M-80. PC-DOS is based on CP/M-80, but not as heavily as CP/M-86. Microsoft and previous authors attempted, with PC-DOS, to make some improvements in moving to the 8086/88, such as:

- Time and date stamping of disk files.
- Changing the name PIP to COPY and making it a built-in command.
- Putting the parameters of COPY and RENAME in the "right" order.
- Allowing disk read and write in blocks

[record sizes] of anything the user wishes, not just 128 bytes as with CP/M. There is still much room for improvement here, but it sometimes will allow reading or writing a file with one call to the system (not counting open and close), instead of having to write 128 bytes at a time in some sort of a loop.

- Showing the exact size of the file in the directory listing.
- The ability to use device names, such as CON: and LPT1: as the "filename" in an FCB, which gives some degree of device independence. Again, there is room for more improvement here, since real device independence is what one would really like.
- The batch facility of PC-DOS is much better than CP/M's SUBMIT, which is a real kludge. For example, the AUTOEXEC facility of PC-DOS allows executing a program or series of programs on power up. We use this at work to read the date and time from a battery-powered clock to set the system's date and time. Doing this under CP/M-86 requires, as far as I know, modifying CP/M's boot file (or giving the appropriate command manually each time the system is booted).

RD: Points granted, with the following reservations: Concurrent CP/M and the new CP/M Plus support time and date stamping of files, and additionally offer the convenience of password protection and "user areas." The new CP/Ms also allow file access in "burst mode" where up to sixteen 128-byte records can be transferred with a single operating system call, but this is still much weaker than the PC-DOS block read/write functions where the record size can be from 1 to 65,535 bytes. Concurrent CP/M has the capability to execute a user-supplied batch file at cold start.

Whether or not PC-DOS's COPY and RENAME parameters are in the "right" order is arguable. CP/M's format (which mimics an assignment statement) certainly seems natural to programmers, but PC-DOS's conventions are easier for new computer users to remember.

In general, I think we will all benefit from the intense rivalry between MS-DOS (and its clones) and CP/M-86. Both systems already show significant improvements over their 8-bit counterparts, and the competition for market share is bound to bring us many additional enhancements in future versions. MS-DOS 2.0, which is scheduled for release soon, incorporates some sophisticated new features including the ability to load user-designed peripheral device drivers without patching the operating system.

IBM PC Character Set Linker

Patrick Banchy, of New York City, wrote in with several suggested modifications to the CLINK utility that was published in an earlier column.

In order to get the program to work at all, when assembled with the Microsoft Assembler to form an EXE file, several pseudo-commands to define segment start/ends must be inserted. Additionally, the instruction:

```
MOV 14 [BX],1024
```

must be modified to:

```
MOV WORD PTR 14 [BX],1024
```

(the Microsoft Assembler does not check the magnitude of the immediate data to determine whether a 16-bit transfer is necessary). On his system, instructions also had to be added to intialize byte 32 of the file control block to zero before the file access, although on my PC this happens automatically when the program is loaded.

Patrick then made some improvements which can be summarized as follows:

All segment registers of a COM program are set to the start of the Program Segment Prefix (PSP), so the first three instructions of CLINK as listed in *DDJ* are unnecessary.

DOS function #37 can be used to set the interrupt vector address.

To decrease the amount of memory made unavailable for use by other programs, he put the code that reads in the file after an ORG 480H statement, and used the default disk transfer address (80H). This results in a bigger load module, but smaller memory loss (1152 bytes for my version vs. 1424 bytes for Patrick's).

Finally, he noted that if CLINK is invoked multiple times, it will reserve a new block of memory each time. This could become quite wasteful, so he changed CLINK to check the contents of the vector location at 007CH. If the link is zero, CLINK reads the character (INT 27H); if the link is nonzero, CLINK points the disk transfer address to the previously loaded table area, reads the new table, and then makes a "normal" exit (DOS function 0).

FLIP Utility for the IBM PC

Simson L. Garfinkel sent in a program listing with the following comments:

"I own an IBM PC with both the color/graphics and monochrome adapters installed. IBM makes no effort to support users who have both of these interfaces installed, other than a brief note on page I-8 of the BASIC manual on how to switch from one display to another.

"I have written a small program (see listing on opposite page) which flips from one display to the other. The program is designed to be converted into a COM file with EXE2BIN program, and for this reason has no stack segment (this generates an error when linked which should be ignored).

"When FLIP.COM is executed, if the monochrome display was being used, the color screen is cleared and becomes the system display, and conversely."

Users who have a high-resolution color monitor will probably want to alter the program slightly so that it selects the 80x25 BW text mode when selecting the graphics interface.

DDJ

OF INTEREST

by Michael Wiesenberg

Mice Are Cheaper by the Thousands

Lisa has a mouse. Xerox has been using mice for years, particularly with SmallTalk. Do *you* need a mouse? The **3G Micromouse** has two pushbuttons and rolls on a table top, moving the cursor on your screen correspondingly. You'll have to do the software interface, but 3G will sell you the hardware for $180, or $72 each in lots of over 1000. **Reader Service No. 101.**

16 Bits for Less

The **Sage II** 68000-based microcomputer with 128K RAM, running at 8MHz and 2MIPS (million instructions per second), with one 640K floppy, and p-System OS costs less than $3600. It's a small 3.9- by 12.5- by 16.7-inch box to which you'll need to add a terminal. Expansion RAM, claims Sage, is at the industry's lowest available price, $250 per 128K, and you can fit another, optional floppy into the box. CP/M-68K is another available option. **Reader Service No. 103.**

Dynalogic's Hyperion is a truly portable, self-contained 16-bit computer that is compatible with the IBM PC running MS-DOS, and costs under $3400. The "entry-level" 18-pound Hyperion has 256K RAM, one 320K 5¼-inch drive (that reads both single- and double-density IBM PC diskettes), MS-DOS, Microsoft Advanced Disk BASIC with graphics, a 7-inch nonglare amber phosphor screen of 80 columns by 25 lines and five pages of display memory, 256 display characters, bidirectional scrolling, soft-key labels on the 25th display line, serial and parallel ports, and a low-profile, detachable keyboard with 84 keys, including ten function keys and numeric keypad and optional audible key click. *Options* include a second floppy drive for $650; a 300-baud, built-in, direct-connect Bell 103J compatible modem with autoanswer and autodial for $495; Dynalogic's IN:SCRIBE text editor for $175; Microsoft's Multiplan spreadsheet for $257; and a soft vinyl travelling case with accessory pockets for $90. These options are standard in the $4995 **Hyperion Plus**, which also includes a time and date clock with battery backup that keeps the time even when AC power is removed. You can also get Microsoft's BASIC compiler, COBOL, Fortran, and Pascal. **Reader Service No. 105.**

What Looks Like an Apple? (But Costs Less)

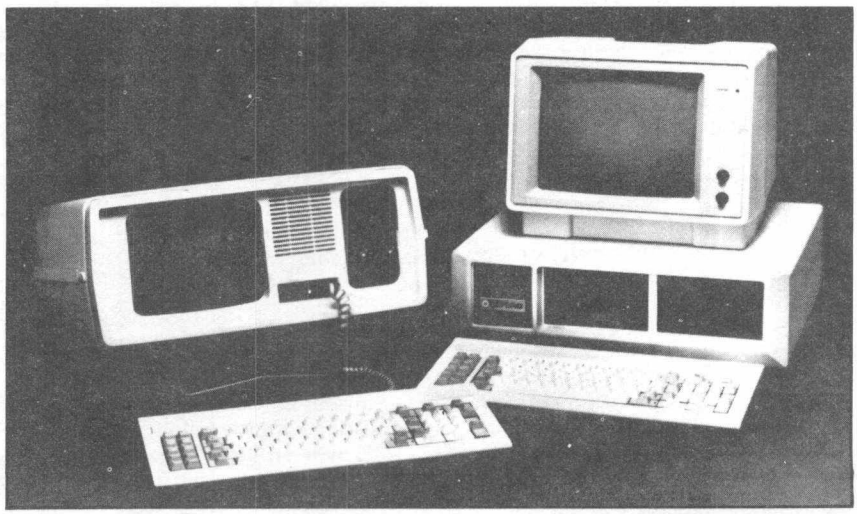
Microcomputer II from **HSC Computer Services** is an Apple-clone for $499 (plus $15 p&h) that runs AppleSoft (and Franklin Ace) software and comes with 64K RAM, 6502 CPU, built-in RF modulator, high-res graphics, cassette, printer, and cartridge interfaces, AppleSoft-compatible BASIC, self-diagnosis cassette, manuals, real keyboard, and switching power supply. Options include printer, joystick, 12-inch green-phosphor monitor, speech synthesizer, Pascal, Forth, and assembly language cartridges, inventory control and accounting cartridges, RS-232C interface, floppy disk drive, and disk controller. **Reader Service No. 107.**

Phone-y Modem

Datamate 103 from **Cermetek Microelectronics** is an intelligent, 300-baud, autodial, full-duplex modem that comes with slim-line telephone receiver, for voice and data communications. It attaches to your computer's RS-232C port, stores the last data number and six other numbers in its nonvolatile memory, makes voice calls even when the modem is turned off, dials from memory, and dials until answered. The package is $295, or modem phone alone for $29.95 and Datamate 103 for $269. They've also got the same package in Bell-212A-compatible, 1200-baud model. **Reader Service No. 111.**

IBM Talks to Phone

I've mentioned **PConnection**, from **Microperipheral**, before: a plug-in IBM PC modem card – direct connect, Bell 103/113 compatible, with autodial (Touch Tone or pulse) – that answers automatically in both originate and answer modes, and fits inside the PC



IBM Clone

Vitek will be distributing **Corona Data Systems'** new IBM-compatible desktop and portable personal computers, that sell for about a third less than the "real thing." You get a green-phosphor monitor with 640-by-325 high-resolution graphics, disk drive, four card slots (3½ in the portable model), 128K RAM (expandable to 512), serial and parallel ports, and detachable keyboard. **Reader Service No. 109.**

cabinet. It used to cost $350. Now they've added automatic disconnect circuitry for failure or carrier loss, and cut the price to $279. They've also added to their line an enhanced version with real-time clock for auto-answer and autodial at preset times, and an additional serial I/O port to increase the PC's communications capabilities, for $350. **Reader Service No. 113.**

Zenith Talks to IBM

Zenith's 3270 Terminal Emulation package for Z-89 and Z-90 desktop computers includes software and bi-synchronous interface card for $750, or software alone for $650. With a printer, the program emulates the IBM 3276 Model 2 with 3278 printer. Zenith's function keys are used in the same way as those of the 3270 series, with color-coded labels included to adapt keyboards to those of the 3270. You need a modem with synchronous RS-232C interface and 64K RAM. **Reader Service No. 115.**

Apple and IBM Are Now Speaking to Each Other

Direct Connect from **Trax** transfers files between Apple II and IBM PC, requiring no extra hardware (beyond a cable that they supply). This software product uses the PC's cassette port and the Apple's I/O, with transfer rates of 10K bps and automatic error checking. A "remote commander" feature controls both machines from the PC, using only one display, and permitting running of Apple programs with standard keyboard input and display on the PC. You also get a Pascal interface for your $170. **Reader Service No. 117.**

Fortran 400 Times Faster

Do you have a CompuPro 8085/88 with Hudson and Associates' 8087 Support Board? If so, you can relink your 8-bit software, including Fortran-80, COBOL-80, BASCOM-80, and MACRO-80, with F87LIB.REL, part of **Avant-Code's Fortran-87**, and run programs up to 400 times faster by taking advantage of the 8087's 80-bit floating-point routines. $200 gets you an 8-inch CP/M disk and manual, and, if you don't have the Hudson board (including 5MHz 8087-3) – installation of which does not void CompuPro's CPU warranty – you can get software and board for $695. **Reader Service No. 119.**

Beat the Computer

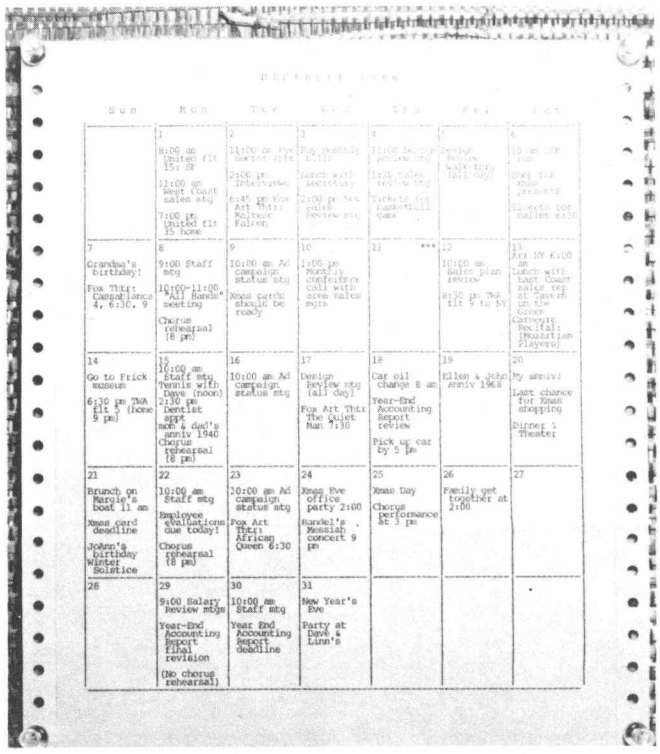
Arisoft's Mike Caro's Video Poker for Apple II includes two games, Jackpot Video Poker and Poker Flurry. The former simulates the progressive jackpot-type video poker slot machines. Although the game is functionally identical to the Las Vegas variety, it contains several added features. Any hand can be replayed. You can set the jackpot to what you like, carry the jackpot over from one playing session to the next, and ask the computer for a mathematical analysis of how you're doing. You can get an instant replay of the last hand. Arisoft describes the feature this way: "Although this won't alter your results or allow you to change your draw, it's good for settling arguments about whether you threw away a pair by accident. It also lets you relive moments of glory." You can change the speed with which the cards are dealt, from superfast to slow motion. The game is accompanied by sound effects, including a victory song when you catch a big hand.

Poker Flurry is a competitive form of video poker. You can play against the computer, or against a friend. Or you can even watch the computer battle against itself! If you do this, you can also learn proper strategy for these casino games that knowledgeable players can get an edge on. (This game is the ultimate decision maker. For example, if you don't know whether to go to work today or play golf, you have the computer battle itself under the names *WORK and *GOLF, and see who wins.) There's even a doubling facility. The disk costs $39.95. **Reader Service No. 121.**

Develop Z8000 on Z80

2500 A.D. Software has a **Z8000 Cross Development** system that runs on any 64K Z80 CP/M system. You get a Z80 to Z8000 translator, 8080 to Z8000 translator, and Z8000 relocating macro-cross assembler. The latter has nested and recursive macros, include files, a linker that links up to 400 files, and relocatable code generation. All this and a 150-page manual for $179.50. **Reader Service No. 123.**

DDJ



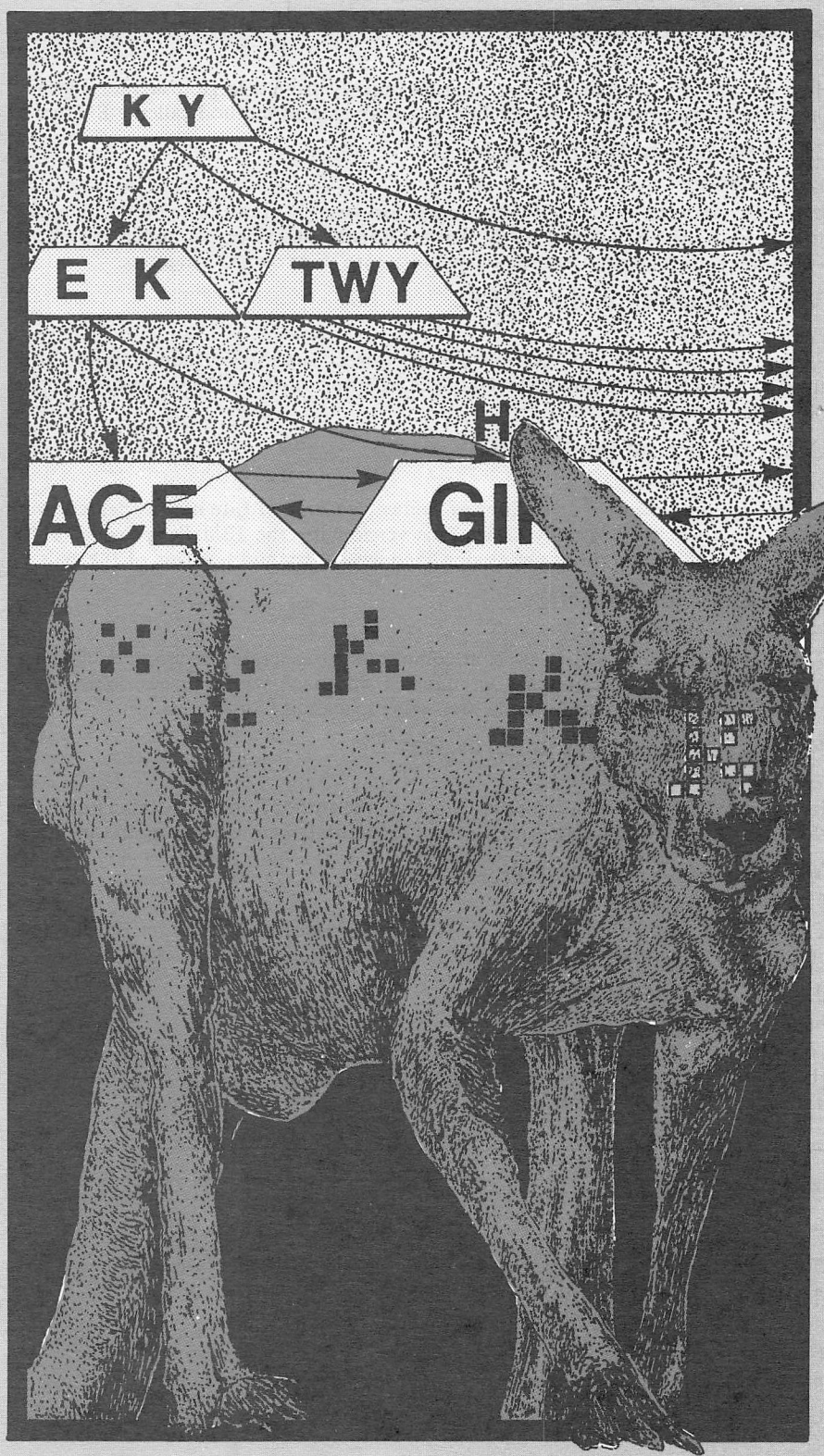
Print Your Own Calendars

Calendar/1 from **Clear Systems** formats calendars for terminal display or printing from text editor entered in no special order or format. It costs $60 and runs under 52K CP/M-80. **Reader Service No. 125.**

#80 June 1983 $2.50 ($3.00 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems



B-Tree ISAM Concepts

CP/M BDOS and BIOS Calls for C

IBM PC: Printing Graphics and The Game of Life

. . . and much more!

Dr. Dobb's Journal
For Users of Small Computer Systems

June 1983 Volume 8, Issue 6

Publisher – Clifford West
Editor – Reynold Wiggins
Managing Editor – Craig LaGrow
Editorial Consultant – Marlin Ouverson
Contributing Editors –
Robert Blum, Dave Caulkins, Dave Cortesi, Ray Duncan, Michael Wiesenberg
Marketing Director – Craig Harper
Marketing Manager – Beatrice Blatteis
Advertising Director – Carl Landau
Advertising Sales – Doug Millison
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Terri Marty, Linda Marohn
Production Manager – Barbara Ruzgerian
Production Assistants –
Alice Hinton, Alana Hunter, Jane A. McKean, Marianne Tryens
Typesetter – Paula Fairchild
Cover Illustration – Al McCahon

Copyright © 1983 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: DeWitt S. Brown, Burks A. Smith, Robert M. Connors, Robert C. Luckey, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Thomas Davis, Jan Steinman, G. Hunter, Ronald E. Johnson, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III, Infoworld, Stan Veit, Western Material Control, S.P. Kennedy, *Ed Moran*, W.D. Rausch. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

CONTENTS

ARTICLES

14 Fast Divisibility Algorithms
by H. T. Gordon
Author Gordon shows how to use the information provided by the digits of large numbers to quickly determine some things about divisibility.

18 B-Tree ISAM Concepts
by Chris Deppe and Alan Bartholomew
How do you retrieve random information quickly from a large disk file? The authors discuss the principles and techniques of an elegant method called a B-Tree.

22 CP/M BDOS and BIOS Calls for C
by Terje Bolstad
Many C compilers for CP/M either omit or severely limit the ability to make direct calls to the BDOS and BIOS. Author Bolstad describes two functions which will allow you to incorporate such capability into your own C programs.

32 Printing Graphics on the IBM PC
by Dan Daetwyler
Earlier versions of PCDOS do not provide direct support for combining a "dot graphics" printer with the Color/Graphics Adapter. Author Daetwyler bridges the gap with this mapping utility.

42 Game of LIFE on the IBM PC
by Simson Garfinkel
While a number of implementations of Conway's Game of Life have appeared over the years, these were usually slow. This version for the IBM PC provides both speed (up to 2.7 generations per second) and easy entry and tracking of the cells.

52 Serial Expansion in Forth
by Wendall C. Gates
Series expansions are widely used in computers to generate precision, non-linear functions. The author presents an unusual approach which places coefficient-exponent pairs into an array, while preserving readability and access.

56 Fast Matrix Operations in Forth, Part I
by Steven A. Ruzinsky
In his search for a fast floating-point language, the author found an ideal in Forth. In this first of a three-part series, he presents some fundamental Forth matrix utilities.

60 Yes, You can Trace Through BDOS
by Do-While Jones
They said it couldn't be done.... Taking up the challenge from our Letters column, the author shows how you can get a trace through your BDOS.

66 Julian Dates for Microcomputers
by Gordon King
Use of a Julian date system allows dates to be stored on computers as integers, but there must be a way to convert between Julian and normal calendar notation. The author presents routines for implementing such a system on your micro.

DEPARTMENTS

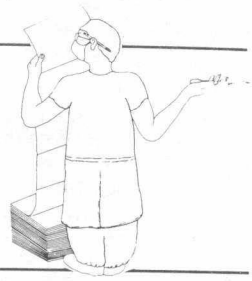
9 Dr. Dobb's Clinic
PC Still Backward, CP/M Plus, sector buffering, the BCB, and more.

72 16-Bit Software Toolbox
8088 Addressing Modes, 8088 Line Generator, PC-TALK III

80 CP/M Exchange
Public domain software, SD-44, and more on CP/M Plus

83 Of Interest

DR. DOBB'S CLINIC



by D. E. Cortesi, Resident Intern

A Yodel from Yap

John Thayer Jensen, of Yap, TT, writes us as follows. "I have the kiss of death: I bought an IMSAI VDP-80 and that put IMSAI out of business. So the first thing is, if there are any IMSAI VDP-80 owners out there who will write me, I can share a partially commented disassembly of IMDOS's BDOS, similar disassemblies of the DIO and VIO ROMs, and can sometimes answer questions about IMDOS and IMSAI hardware.

"I also bought a now-obsolete multitasking operating system called FAMOS from MVT Microcomputing. They never sold any more (as far as I can tell) and while they make a token effort at support, they can't be expected to do much – and the thing doesn't work. If there are any FAMOS users reading this, I have next to nothing to offer, but an infinite number of questions to ask someone!

"It would be wonderful to hear from some fellow VDP-80 and/or FAMOS users. I can't check into CBBSs, can't go to meetings, can't phone anyone. We have a U.S. zipcode, proving that we are not at the end of the world (although on a clear day, you can see it from here), and domestic first-class mail will reach us by air."

If you would like to correspond with Jensen in his lonely outpost (Yap is an island in the mid-south Pacific), you can write to him at P.O. Box 358, Yap, TT 96943.

PC Still Backward

In March we expressed our puzzlement at the results of printing an ASCII backspace from BASIC in the IBM PC. The result of

```
PRINT CHR$(8)
```

is precisely nothing, while the result of

```
PRINT "wotizzit:",CHR$(8)
```

is (as we described it) a small reverse-video diamond which we couldn't find in any manual.

Several readers responded. Randolph Fritz of Mahwah, New Jersey and Bob Taylor of Buffalo, New York wrote to point out that the complete character set is shown on pages C-12 and C-13 of the *IBM Technical Reference,* and it includes the one we couldn't find. Robert Pirko of New York added that "The funny character you see is intended to be a reverse-video *circle.* For the graphics adapter in the 40-character mode, it looks roughly like a circle. In the 80-character mode, the width is cut in half and the circle does look more like a diamond."

David Kellogg, also of New York, called to say the same, and added that if you print character 226, the *Technical Reference* says you should get a lower-case gamma, but you actually get an upper case gamma.

OK. The whole situation is very confusing, but we think we understand it now. Follow us through this. (1) IBM, for whatever reason, implemented a graphic symbol for every byte value from 001 to 254 inclusive, leaving only 000, 032, and 255 as blank or null displays. The bytes from 032 to 127 have their ASCII symbols; the others produce unique special characters. (2) The PC's ROM implements a simple dumb-terminal function (WRITE-TTY) which takes ASCII bytes and displays them. It implements the control characters as a teletype would, including carriage return (CR), line feed (LF), and backspace (BSP). However, WRITE-TTY does *not* implement cursor addressing (as any modern terminal should, no matter how dumb).

But (3) the BASIC interpreter needs direct cursor addressing for its screen editing, so it calls upon the more primitive routines in ROM. These routines supply cursor positioning and screen output (SET-CPOS and WRITE-AC-CURRENT). Since they are not trying to emulate a terminal, they treat ASCII control characters as data. A control character like BSP, sent via WRITE-AC-CURRENT, will go into the screen display as a symbol; it will not have its standard ASCII effect.

Then (4) the BASIC interpreter also used the primitive ROM functions to implement the PRINT statement; thus any character written via PRINT will go straight to the screen as a symbol.

Except that (5) the interpreter wants to supply some modicum of standard control characters, so it intercepts BEL (and toots the horn), CR (and returns the cursor), TAB (and tabs by 8), LF (and moves the cursor down), and FF (and clears the screen). In effect, it does the job of WRITE-TTY, but in the interpreter code. It *should* intercept BSP as well; it apparently does so when the cursor is at the left margin, but not otherwise. So a backspace at the left margin acts like a backspace (doing nothing), but once away from the margin, it gets through and becomes a symbol. This appears to be a bug in BASIC.

Finally (6) the interpreter wants users to be able to move the cursor freely, so not only does it implement the LOCATE verb, it *also* intercepts the ASCII control characters US, RS, FS, and GS and treats them as commands to move the cursor in the four cardinal directions.

The net: a BASIC programmer has *two* ways to position the cursor (neither compatible with any other BASIC), but lacks both a standard backspace and the nonstandard symbols of bytes 028-031. Meanwhile, the user of another high-level language can use the backspace (since most of these use WRITE-TTY) but cannot move the cursor. This is how your major corporations do whatcha call your systems analysis and design stuff, see?

SuperKludge

Nick Hammond, now of Torrens, Australia, has sent us a lovely kludge for CP/M 2.2. Here's what he says.

"A couple of months ago, I was asked to modify an 8-inch, single-sided, single-density CP/M system to allow a larger-than-standard directory. It proved to be a fairly simple mod and I thought you might like to share it with your readers.

"A standard 8-inch, single-density CP/M diskette has a file directory that will hold 64 entries. Since each 16K extent of a file requires one entry, this may amount to less than 64 actual files. Given the maximum data storage of 240K odd, this is a reasonable number, but occasions will arise when we need more. The example that prompted this note was a need to fit a dBase II application with a large number of command files onto one disk, leaving the second free for data.

"Fortunately, CP/M 2.2 is both flexible and well thought-out, and expanding the directory can be done relatively simply. Two things must be done: generation of a diskette which will look okay to both the standard and modified systems, and insertion of a two-byte patch to modify the BIOS for the new directory size.

"The standard directory on a single-density 8-inch disk occupies the first two 1K blocks after the system tracks. The modified directory will occupy the first four, and we therefore need to reserve the third and fourth blocks. If this is not done, the directory could be overwritten when used with an unmodified BIOS, and would then appear full of garbage when used with a modified one.

"We can reserve the extra directory blocks by making the first file written to disk a dummy, full of E5 bytes. To the standard system, this will appear as a normal file; to the modified system, the E5 pattern will look like two blocks of empty directory space. To ensure that the dummy file occupies the third and fourth blocks, it must be written under a standard system and it must be the first file on the diskette. If your SYSGEN procedure copies hidden files to the new disk, the dummy file must be written before you SYSGEN the disk. The following DDT session illustrates the procedure.

```
A>ddt
DDT VERS 2.2
-f100,900,e5          fill 2K with E5h
-g0                   exit DDT
A>save 8 b:empty      save dummy file
A>stat b:empty $sys   and hide it
```

"CP/M gets its information on directory size from the Disk Parameter Block (DPB), a data area contained in the BIOS. To enlarge the directory it is necessary to alter two bytes in the DPB. The first is DRM, a count one less than the number of directory entries. The second is ALLOC0, a bit-map of the disk blocks reserved for the directory.

"The DDT session that follows shows how to find the DPB and how to make a small command file, KLUDGE.COM, to alter it. The DPB that this procedure locates is the one for the default drive. In most systems there is only one DPB for each disk format and the modification will affect all drives. If it doesn't on yours, just repeat the exercise with a different drive logged in.

```
A>ddt
DDT VERS 2.2
-a100                 assemble a program
0100 mvi c,1f         to locate the DPB
0102 call 5
0105 rst 7
0106 .
-g100                 execute it
*0105

-x                    HL has the address
C0Z1M0E1I0 A=14 B= ... H=FE14 ...
-hfe14 e              calculate its size
FE22 FE06
-dfe14 fe22           dump it to make sure
FE14 1A 00 03 07 00 F2 00 3F 00 C0 ...
FE20 00 02 00 ...
-a100                 assemble a program
0100 mvi a,7f         to patch the DPB
0102 sta fe1b         ...change DRM to 127
0105 mvi a,f0         ...reserve 4 blocks
0107 sta fe1d         ...in ALLOC0
010A ret
010B .
-g0
A>save 1 b:kludge.com
```

"After running KLUDGE, the directory will have space for 128 entries, and should stay that way until the next cold boot, which overwrites the BIOS. Some manufacturers such as Osborne have chosen, for reasons best known to themselves, to rewrite the BIOS at warm boot also, contrary to CP/M specifications. On these systems, KLUDGE will have to be run after each boot." [*A BIOS that supports multiple formats may rebuild the DPB when a disk is selected after a warm start – DEC.*]

A Stack of Boots

Aubrey Hutchison has checked in with a warning for users of the California Computer Systems BIOS supplied with the 2422 disk controller board. "After running for 18 months," he writes, "I found a problem that was bothering me from time to time. At times when using PIP, the machine would appear to hang up; at other times it would seem to PIP forever; but most of the time PIP worked as expected.

"The problem turned out to be related to the CCS boot code and boot ROM. The stack pointer is set by the ROM to be the top of RAM – 56Kb in my case, since at boot time the top 8Kb of RAM is banked out. When the ROM enters the cold-boot routine loaded from disk, it leaves the stack in the same place. The cold-boot routine doesn't change it. The CCS BIOS did not set the stack pointer either [*it sets it on a warm boot but not a cold one – DEC*] so until the CP/M CCP took over, the stack was not set at some other location. Since the BIOS used the stack for one or two (maybe three) levels, the stack was having a good time playing around in the BDOS. The damage must have been small since I used it in this condition for 18 months.

"My method of correcting the problem was to fix the location of the stack at the entry to the cold-boot routine. I found space to insert

```
lxi h,0100h
sphl
```

at the start of the cold-boot code without changing the original code. The same thing could be done in the BIOS [*at label BOOT*]."

What do you suppose the BDOS has in its highest two or three words that overwriting them would cause trouble, but only rarely?

The Intern's 2.2 BIOS

In the last couple of columns we talked about the fancy BIOS we built for a CP/M 2.2 system with banked storage. We have CP/M 3 running now, and in building its BIOS we had to pretty much rip up the new 2.2 BIOS and lay it down again. It seems a shame to let such a pretty thing die so soon, but we really have no more use for it. And who else would, unless they had exactly our hardware configuration? Or unless they had a masochistic desire to read approximately 95 pages of heavily commented Z80 assembly code. . . .

If you have such a masochistic desire, you can have a copy of our 2.2 BIOS for your bedside reading. Send your own 8-inch, single-density diskette and a sturdy self-addressed, postage-paid diskette mailer, and we'll duplicate the source files onto it. Address the package to "The Intern's 2.2 BIOS" c/o PCC, P.O. Box E, Menlo Park, CA 94025. Note that you should not expect to actually *use* this code. If you have any idea for doing so, be aware that it needs RMAC and LINK; it comes with minimal documentation; and it carries *absolutely no warranty or support of any kind.*

CP/M Plus – Sector Buffering

The CP/M 3 (or CP/M Plus) BDOS buffers disk sectors in storage in an attempt to speed up processing. The attempt may be successful in some cases (we haven't tried any direct-access I/O yet), but the BDOS uses its sector buffers in a way that is far from optimal for sequential I/O. This is demonstrated by some experiments we made recently.

A CP/M 3 BIOS for a banked system supplies the address of two words in storage in the Disk Parameter Header that it returns from a SELDSK call. These are the anchors of two chains of Buffer Control Blocks (BCBs). Each BCB describes a buffer that the BDOS may use to save a disk sector (a physical sector, not a 128-byte logical sector). There are separate chains for directory and data sectors. According to the *CP/M 3 System Guide*, page 8, "In a banked environment, CP/M 3 maintains a cache of deblocking buffers and directory records using a Least Recently Used (LRU) buffering scheme." And on page 46, "In general, you can enhance the performance of CP/M 3 by allocating more BCBs."

Always anxious to improve performance, we gave CP/M 3 a grand total of 71 sector buffers, each one a kilobyte, spread over our four-bank system. Seventy-one kilobytes of data space (plus another 8K of directory buffers) is not exactly a SemiDisk, *but it's a lot more buffers than*

(Continued on page 89)

Fast Divisibility Algorithms

by H. T. Gordon

H. T. Gordon, College of Natural Resources, University of California, Berkeley 94720.

It is well known that the lowest digit of an integer, if not 1, 3, 7, or 9, suffices to establish divisibility by 2, 5, or 10. Algorithms for divisibility by other small integers (such as 3, 7, 9, or 11) are not common knowledge. They require information from *all* the digits of the number, and are too complex to be part of elementary arithmetic but too trivial for the mathematical elite. Even the "obviously divisible" integers contain useful information in the next-to-last digit. E.g., numbers ending in 5 are divisible by 25 if the preceding digit is 2 or 7, while numbers ending in 0 are divisible by 25 if the preceding digit is 0 or 5. Numbers ending in 2 or 6 are divisible by 4 if the preceding digit is odd – if it is even, the number has only a single factor of 2; the opposite is true of numbers ending in 0, 4, or 8.

The algorithm for divisibility by 3 or 9 is relatively simple. It requires information from *all* the digits, but is independent of the *order* of the digits. It calculates the "reduced digit-sum" by adding all the digits; whenever this becomes 10 or more, 1 is added to the low digit and the high digit is discarded so that the sum is always a number from 1 to 9. If the final reduced sum is 9, the number is divisible by 9. If it is 3 or 6, the number has a single factor of 3. E.g., the number 123456789 has a reduced sum of 9 and so is divisible by 9. This is true of all the possible permutations of these digits, and also true for 12345678 and all its permutations. However, 1234567 has a reduced sum of 1 and so cannot be divided by 3, nor can any of its permutations. The presence of zeroes within the numerical sequence does not affect the truth of the algorithm; e.g., 45, 405, 4005 all are multiples of 9, and so of 45.

The algorithm for divisibility by 11 is somewhat more complex, since it depends on the *order* of the digits (only a fraction of the possible permutations being multiples of 11). It requires the calculation of *two* digit-sums. The "high" sum is that of the first, third, fifth, etc. digits; the "low" sum is that of the second, fourth, sixth, etc. digits. Within each of these sets, however, the order of the digits does not affect the truth of the algorithm since this requires only that the *difference* between the two sums be a multiple of 11, or 0. The difference will always be 0 if each sum is reduced by 11, though it is simpler to reduce only the difference, in one step, if it is not 0. An example is the number 12435687 with both high and low digit-sums equal to 18, and so divisible by 11. If one *adds* the two sums, the value 36 reduces to 9, so the number is divisible by 99. Which set is "high" does not matter; e.g., 21346578 is also divisible by 99, and of course also by 2. Though more complex, this algorithm can test divisibility by 3, 9, or 11 in one operation. For gigantic numbers, the digit-sums can be reduced by 99 since this is a multiple of both 9 and 11 – if a sum reaches 100 it reduces to 1, etc.

These algorithms are valid for number bases other than decimal, although this alters the factoring integers. In octal, successive multiples of the highest digit (7) are 16, 25, 34, 43, etc., so that numbers of any size will yield a total reduced sum of 7, while successive multiples of its base (8) + 1 (= 9) are 11, 22, 33, etc. In hexadecimal (base 16), the total reduced sum of numbers factorable by its highest digit (F, = 15 decimal) must always be F (e.g., 1E, 2D, 3C, etc.), while the "elevenish" series (11, 22, 33, etc.) has all multiples of decimal 17.

As in the decimal "base minus 1" algorithm, in which reduction by 9 reveals multiples-of-3 when the reduced sum is 3 or 6, so in hexadecimal does a reduced sum of 3, 6, 9, or 12 reveal divisibility by 3 (but not 5), while a reduced sum of 5 or A (decimal 10) reveals divisibility by 5 (but not 3). Only in decimal do multiples of 5 always have 0 or 5 as the lowest digit. The general rule (at least for bases that are even numbers) is that numbers divisible by the digit that is half the base always end in 0 or that digit.

Divisibility by 7 can now be seen to require a general "base minus 3" algorithm, that in octal would recognize divisibility by 5 and in hexadecimal divisibility by 13. I have only studied the algorithm for decimal notation. It is even more complex than divisibility by 11, requiring not only calculation of a similar high and low pair of digit-sums, but positional "weighting" of each digit. The *order* of the digits in each set is significant. Relatively few permutations within each set retain divisibility by 7, since only digits with the same weighting factor can be interchanged. Weighting involves multiplication by 1 (i.e., no change) for the first, fourth, seventh, etc. digit (reading the number from left to right), multiplication by 2 for the second, fifth, eighth, etc. digit, and multiplication by 4 for the third, sixth, ninth, etc. digit. However, digits can be reduced by 7, to a pseudo-heptal notation; i.e., 7 becomes 0, 8 becomes 1, and 9 becomes 2. Also, the product of a digit by its weighting factor can be reduced by 7; e.g., if the digit is 6 and its factor is 4, the product ($24 = 3 \cdot 7 + 3$) can be replaced by 3. Since such repetitious calculation would slow the algorithm down, the operation of replacing each digit by its weighted and reduced digit can be done by the following look-up table:

Decimal Digit to be "Replaced"

Order	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	0	1	2
2	2	4	6	1	3	5	0	2	4
3	4	1	5	2	6	3	0	4	1

For example, the algorithm alters the number 421589637 to 444521660, which has 16 for both its high and low digit-sums. The original number is therefore divisible by 7 since this requires that the difference be either 0 (as in this case) or also a multiple of 7. Since this number also happens to be divisible by 9, it is divisible by 63. Another example is 10101, which alters to 10402 with a high digit-sum of 7 and a low of 0, revealing divisibility by 7 and (since the total unaltered digit-sum is 3) by 21.

All these divisibility algorithms involve no actual division. They only determine whether division by the small prime being tested is possible, and can do this relatively quickly even for gigantic numbers by reducing them to much smaller numbers that retain the same divisibility. A high proportion (56.6%) of very large random numbers ending in 1, 3, 7, or 9 will be divisible by 3, 7, or 11 and so can be proved to be non-prime. Algorithms for divisibility by higher small primes (13, 17, 19, etc.) could surely be devised, but would "sieve out" ever-decreasing fractions of non-primes at an ever-increasing cost in complexity. However, one never knows when exotic algorithms may prove useful, if only as clues to far more powerful ones.

One human-interest element is that I did a casual search for these algorithms in various math books, in the certainty that

they were known, but failed to find them. Rediscovering the 3-6-9 and the 11 algorithms – by simple inspection of the digit sequence of small multiples – was child's play. Subsequently I found these algorithms on pages 25-27 of the *VNR Concise Encyclopedia of Mathematics*, recently published (or reprinted) by Van Nostrand Reinhold. This is the work of East German mathematicians and presents a few uses in detection of errors in complex calculations. It omits the divisibility-by-7 algorithm, though I'm certain this was also discovered (probably more than once) long ago. I found it less obvious than the simpler algorithms. Its computational drudgery would, in the pre-computer era, have discouraged its use. Even the extension to other number bases has probably been done before, though like other things the elite see as trivial, it has been forgotten! The VNR Encyclopedia reference pointed out something I had ignored, that what I've called the "reduced sum" is not useless even when non-divisibility is found, since it is then the remainder that would be obtained if the division were actually done. Perhaps the divisibility-by-3 algorithm could be used to test random-number generators, since 1/3 of a fairly large sample of large numbers ought to be divisible by 3, 1/3 should have a remainder of 1, and 1/3 a remainder of 2. ▶▶J

B-Tree ISAM Concepts

by Chris Deppe and Alan Bartholomew

How do you retrieve random information quickly from a large disk file? Every programmer who has written software for business has had to deal with this problem. Fortunately, computer scientists have refined an elegant and simple method for retrieving information called a B-Tree. The objective of this article is to explain the principles and techniques involved in a B-Tree.

Why ISAM?

With current hardware technology, about the only way to quickly retrieve information out of a disk file is using an indexing method. There are many methods of indexing data. ISAM, Indexed Sequential Access Method, has become very popular for use in many business systems. It allows the information to be retrieved randomly by data value (indexed) and in sorted order (sequential) using just one index file. This dual ability is what distinguishes ISAM from other methods, such as Hashing. While there are many ways to implement an ISAM index, the B-Tree is generally accepted as being the current "state of the art."

General ISAM Principles

An ISAM file is a file composed of individual pieces of information called "keys." A key is an ASCII string representing some value in a data record. The index is arranged in such a way that keys can be retrieved randomly and sequentially, along with their associated data record numbers. There are a number of different schemes designed to serve this purpose, a B-Tree being only one. To better appreciate the B-Tree, we will look at an older method, called the binary tree.

A tree structure is called such because if all the search paths are drawn out, they resemble an inverted tree as shown in Figure 1 (at right). The search starts at the root and progresses down the tree until it possibly reaches the bottom (called leaves, or leaf level). Note also in Figure 1 that from any point in the tree, there are two paths to the next lower level, hence the term "binary tree." In a simple binary tree, each key is stored in an individual record, or node. Each node also has two pointers to other nodes. These pointers are what make up the search paths through a tree.

A search through a binary tree is, on a basic level, a simple procedure. To find the key "G," it is first compared to the key in the root node. If "G" matches, then the search is successful. Otherwise, a decision must be made, that is, where to go next. If "G" is smaller than the root node, then the path to the left is taken. If "G" is larger than the root, then the path to the right is taken. The path is followed and the new node is read. "G" is then compared to the key in this node as before, and the appropriate action is taken. The search continues until either "G" is found, or a leaf node is unsuccessfully evaluated.

As long as the nodes are kept in memory, the binary search is an efficient method. Once the nodes are stored on the disk, however, the performance quickly degrades because of the large number of disk accesses required.

Other problems can also arise when keys are inserted. This can have the effect of making the tree unbalanced – that is, some search paths being longer than others. This requires that the search and insertion algorithms be aware of this possibility, and subsequently these procedures become much more complicated.

The Basic B-Tree

The problems with the binary tree in large disk-based filing systems gave an incentive for researchers to look for something better. In the late 1960s a number of people independently designed such a method called a B-Tree. Apparently the reason for the name, B-Tree, seems to be the fact that R. Bayer, then at Boeing Scientific Research Labs, was one of the pioneers of the method ("Bayer-Tree").[1]

Figure 2 (page 19) shows an example of the B-Tree format. The most obvious difference between a binary tree and a B-Tree is that from any one node there can be more than two paths to the next node. This has the effect of allowing many more keys in each node of a B-Tree than with a binary tree. For instance, a binary tree with ten-byte keys, holding one million keys, could take as many as twenty node searches to find any one key. A B-Tree with the same conditions (assuming ten keys per node) would at the most take five node searches. In the case where node searches correspond to disk accesses, the search time difference is obviously dramatic.

Searching a B-Tree

To find a key in a B-Tree, the first step is to look at the root node. In the

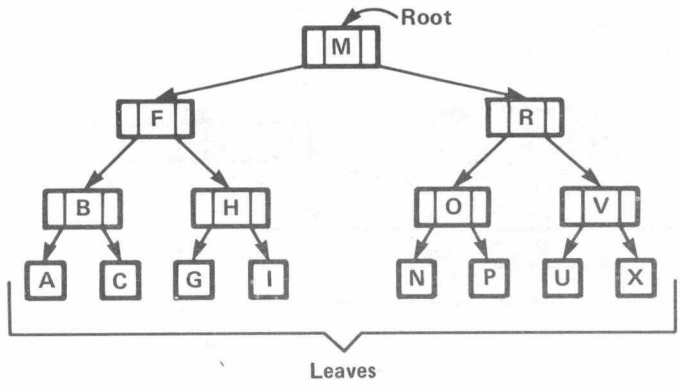


Figure 1. A Binary Tree

Copyright © 1983 by Business Computing Press.

Alan Bartholomew, Business Computing, 2210 Wilshire Blvd., Suite 289, Santa Monica, California 90403.

Chris Deppe, 6333 Canoga Avenue #191, Woodland Hills, California 91367.

example shown by Figure 2, there are a maximum of three keys in each node. To decide what to do, a process called scanning is used. This involves sequentially looking at each key in the node (note they are stored in ASCII sequence) and stopping when either a matching key is found (in this case, the search is successful, and it ends) or a higher key is found. If no match is found, then a decision must be made.

Take, for example, a search for the key "L." By scanning the root node in Figure 2, we find that there is no matching key. We therefore must follow a pointer to the next lower level. The pointer to follow is the one which sits where the key "L" would be if it existed in the node. In this case we would follow the middle pointer to the next lower node. This node is then read and the same procedure is followed. This continues until either the key is found, or a leaf node is unsuccessfully scanned.

Inserting into a B-Tree

Note that since we must determine where the key would sit if it existed in the node, we now have the needed information to insert the key. Insertion into a B-Tree uses the search already described. Then the key is simply inserted into the node where it should be. Note that keys are always inserted into leaf nodes (an insertion depends not on finding the key, but on finding a place for it). Since an unsuccessful search always ends at the leaf level, then all keys will be inserted there.

If the node into which the key is to be inserted is full, then a split occurs. The original node and the new key are divided up into two new nodes. Since there is now a new node, a pointer to the new node must be inserted into the level above. Usually the middle key of the two new nodes is brought up to the next level to be used as a separator. If the node above is also full, then it too might be split. This can continue up to the root. If a split of the root node occurs, a new root node is created so that the tree becomes one level higher.

A B-Tree by nature is always a balanced tree. Since all node expansions are done on the same level, the tree never gets unbalanced. An insertion will never increase the search path to one leaf node and not the others.

Deletion of a key is simply finding the key and taking it out of the node. If a key doesn't reside in a leaf, then a new key must take its place to provide the same paths as the deleted key. This new key is found by getting the next key in sequence from the deleted key.

As stated before, the search time in a B-Tree is much more efficient than in a similar binary tree when stored on disk. Sequential processing in a B-Tree is not so easy, however. Because the keys are distributed between all levels of the tree, keys must be retrieved by following the links up as well as down the tree, working from the left to right. This becomes awkward and slow. Try it by hand using Figure 2 and you will quickly see why.

If the sequential processing of a B-Tree could be improved while retaining the random search efficiency and balanced nature, we would have a much better method.

The B+Tree

This brings us to one of the most important variants of the B-Tree called the B+Tree. Figure 3 (page 21) shows an example of the B+Tree format. In a B+Tree, all the keys are stored in leaf nodes. The upper levels simply provide pointers to the next lower level, and so on until the leaf level is reached. In addition, all the leaves are linked together. What we have, in effect, is a tree which provides a B-Tree type of path to the proper position in a sequential list of the keys.

Since all the keys reside on the leaf level, sequential processing is now very easy as each key is linked together sequentially in the leaf nodes. This gives the B+Tree the desired additional characteristics of an ISAM while retaining the simplicity of a normal B-Tree.

The B+Tree search uses the upper levels as a roadmap to the next level, and it is only until a leaf is reached that the key is actually looked for. Therefore, all searches use the same number of node reads. Although this guarantees that every search will be a "worst case" of a normal B-Tree, we have seen that this worst case is very good under most circumstances. In addition, since all searches take approximately the same amount of time, a high degree of consistency is achieved, which can have its benefits in a real-world situation.

One nice benefit of the B+Tree structure is that since the upper levels are just pointers, a delete doesn't have to worry about revising pointers (deletes only operate on leaves which have no pointers).

Key searching and insertion methods are basically the same as in a normal B-Tree. The specific B+Tree we implemented uses a scheme which uses the high key from the left node to provide a separator (located in the index node above) between the left node and the right node. For example, in Figure 3, a search for the key "H" would produce the following. First, "H" is compared to the first key in the root node. Since "H" is less than "I" the search proceeds by reading the next

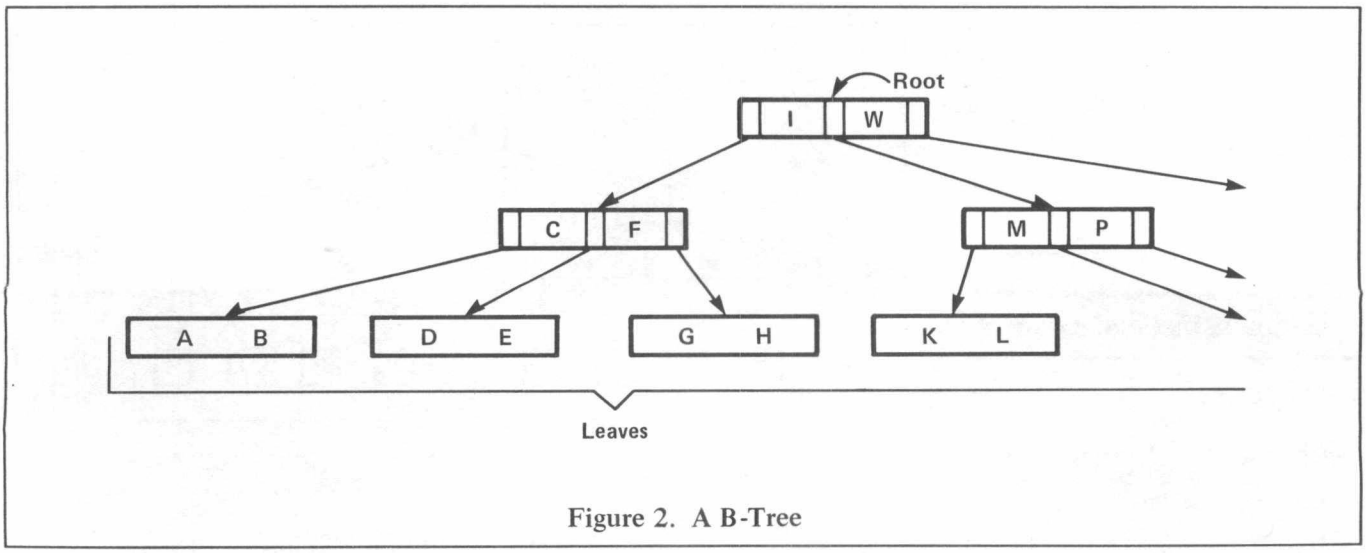


Figure 2. A B-Tree

lower node using the left-most pointer. Then "H" is compared with the first key in the lower node. Since "H" is not smaller than or equal to "C," we must look at the next key. Again, since "H" is not smaller than or equal to "F," we move on to the next key. The next key is "I." "H" is smaller than "I." Since "I" represents the largest value stored in the node its left pointer points to, we follow that pointer.

Splits also use this same high-key logic. At split time, the high key is taken from the left node and it is then inserted (with a pointer to the right node of the split) in the index node above (see Figures 4a and 4b on page 21).

One thing should be mentioned about the B+Tree. Since all the keys are in leaves, the nodes above represent an additional overhead not found in the basic B-Tree, which uses only key values in its tree. With the advent of inexpensive mass storage, though, a slight bit more storage space taken up is well worth the many benefits a B+Tree has to offer.

B-Trees on Micros

Many implementations of the B-Tree ISAM have been created. For example, IBM uses a variant of the B-Tree for their mainframe computers called "VSAM." More recently, the B-Tree ISAM techniques have been applied to microcomputer applications. For example, dBase II uses a B-Tree method for indexing. Also, a number of companies have developed B-Tree subroutine products for microcomputers. These subroutines can be added to an application being developed. The application programmer uses call statements to the indexing functions built into the B-Tree subroutine without having to be concerned with all the details of how it works.

B-Tree implementations have been produced to support a wide variety of languages. The authors have developed a version written in Forth. Other languages supported by various B-Tree packages include BASIC, Pascal, C, COBOL, Fortran, and others.

For the programmer interested in creating a B-Tree "from scratch," the references for this article will be helpful. These references provide more details about B-Tree techniques and the many subtle variations possible in designing a B-Tree ISAM.

The authors have implemented B-Tree ISAM technology in a Forth utility called INDEX+. Written for Laboratory Microsystems Forth, the package includes fully documented source code and a manual which includes tutortials on B+Tree technology and use of the utility. For more information, contact Laboratory Microsystems.

References

[1] Knuth, Donald E. *The Art of Computer Programming, Volume 3, Sorting and Searching,* Addison-Wesley Publishing Company, 1973.

[2] Comer, Douglas, "The Ubiquitous B-Tree," *Computing Surveys,* Volume 1, No. 2, June 1979.

DDJ

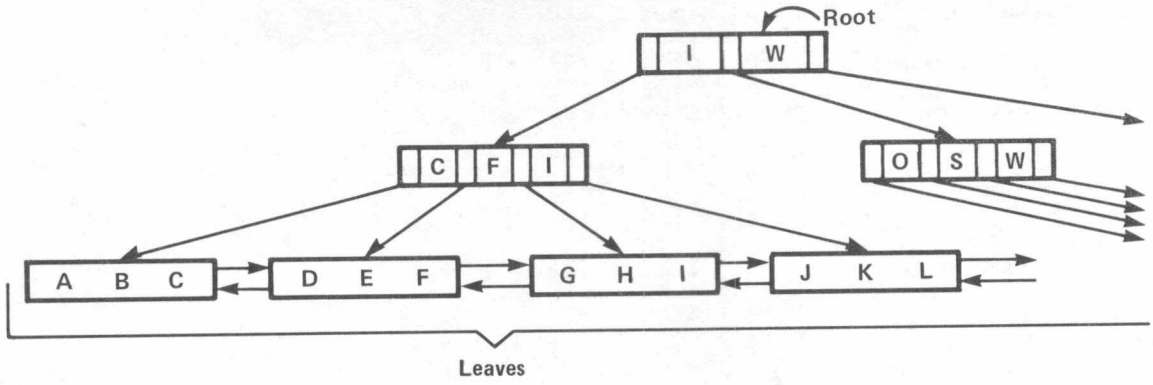


Figure 3. A B+Tree

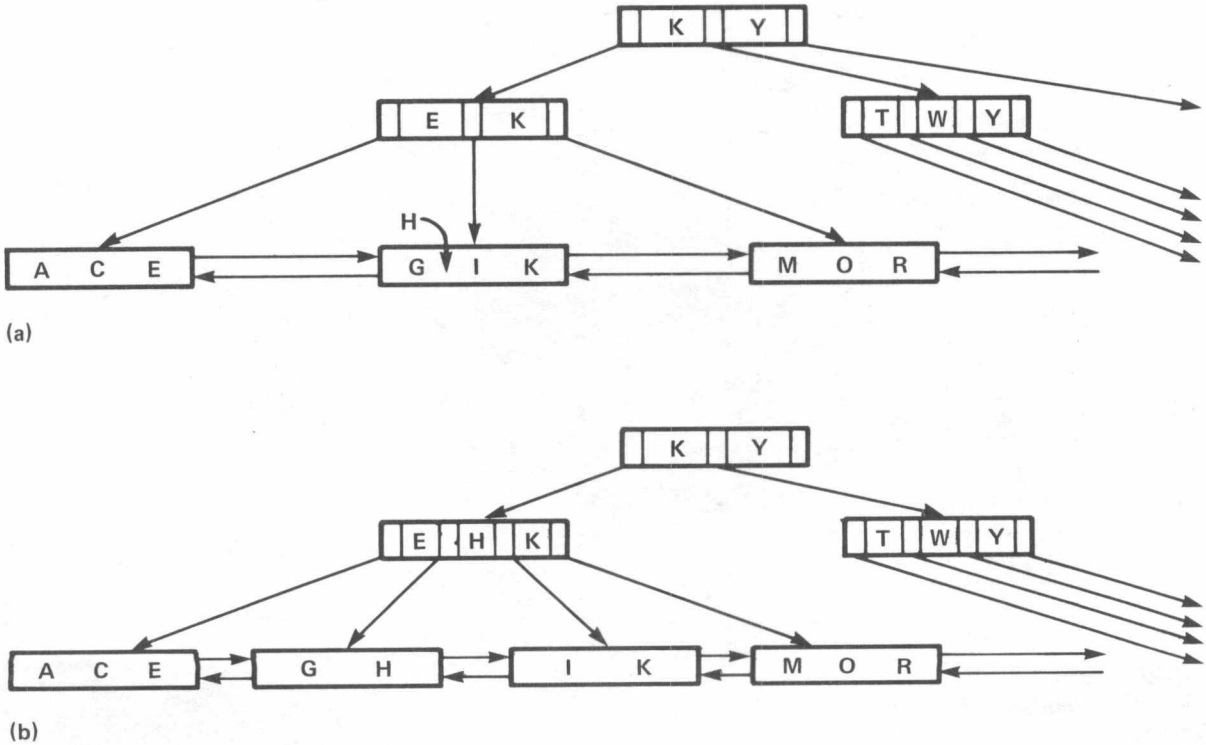


Figure 4.

Figure 4(a): In order to insert the key "H," the mode to receive it must be split. Figure 4(b): The tree after the split. Note that the new pointer in the upper node is the high key from the left node of the two new nodes.

CP/M BDOS and BIOS Calls for C

If you should want to write CP/M utilities in C, you will probably need facilities to access the BDOS (Basic Disk Operating System) and/or the BIOS (Basic Input/Output System). Many C compilers for CP/M do not include such functions, and several of those which exist have severe limitations, in that they may not return proper values in all cases.

The two functions bdos() and bios() described here will enable you to incorporate direct BDOS and/or BIOS calls in programs written in C. Please note that programs that call bdos() or bios() will not be portable beyond CP/M and the 8080 and Z80 series CPUs.

bdos() and bios() were originally written for the C/80 compiler from The Software Toolworks, but they will also compile/assemble/work with other compilers which push arguments on the stack in a non-reversed order before calling a function. Such C compilers include Small-C and most of its derivatives. bdos() and bios() have been tested to work properly with both The Software Toolworks C/80 and The Code Works CW/C compiler.

bdos( )

The bdos() function sets machine register C to the function number specified in funct, and the register pair DE to the value given in arg, and initiates a call to BDOS. The function number may be specified numerically or symbolically. No checking of the arguments is done.

If funct is either RETVN (12), RETLV (24), GETAA (27), GETROV (29) or GETDPA (31), bdos() returns the value remaining in register pair HL on return from BDOS. For all other BDOS functions, bdos() returns the value in A, with sign extension. In this way, BDOS error (255 or FF(hex)) is returned as -1.

bios( )

The bios() function sets machine register pair BC to the value given in arg1, and the register pair DE to the value given in arg2, and initiates the appropriate BIOS call by transferring control to the BIOS jump vector entry point specified in funct. This entry point may be specified numerically or symbolically. No checking of the arguments is done.

If funct is either SELDSK (9) or SECTRAN (16), bios() returns the value remaining in the HL register after execution of the BIOS call; otherwise it returns the value remaining in register A on return from BIOS, with sign extension.

Note that you must specify all three arguments in the bios() function. If you do not need the last one, you must set it to 0 or any other value.

Installation

You may type the source code in Listing 1 (page 24) into a file called CPMCALL.C. To use bdos() or bios() in a C source file, you must #include "CPMCALL.C" in that file.

A simple example on how to use bdos() and bios() is shown in Listing 2 (page 27). In this example, bdos() is used to get the current disk and bios() is used to print it on the console. You may use this example to test that you have installed bdos() and bios() correctly on your system. When run, the program should write the uppercase character corresponding to the current (logged) disk drive on the console. Log onto different drives and check that the program writes out the correct letter.

Normally it is not advisable to do input/output via bios() or bdos() (as done with bios() in the example), if this can be done via other conventional C functions, such as putchar().

In order to be compatible with code for other compilers which do offer the same functions (e.g., SuperSoft's C compiler), you should refer symbolically to the bdos() and bios() function numbers (as shown in the listings) and never use numbers directly. The reason for this is that other compilers may not use the same numerical values for bios() function numbers as used in this version of bios().

You may want to delete (or "comment out") all the #define macros which will not be used, so that they will not occupy unnecessary space in the compiler's macro substitution table.

After compilation, bdos() and bios() will assemble under CP/M's ASM, The Software Toolworks' AS, and most other 8080 assemblers. Both functions may be assembled for nonstandard versions of CP/M by changing the CPBASE equate from 0 to the required value.

If you should want to delete the bdos() function and make a separate function of bios() only, you need to include the CPBASE equate in front of the bios() assembly code. To use bdos() and bios(), refer to the CP/M Interface Guide and the CP/M Alteration Guide.

A final warning: You should avoid direct CP/M calls whenever possible. Even though they enable you to do wonderful things in CP/M, they will limit the portability of your programs.

The code may be copied, used, and distributed by anyone, commercially or otherwise, provided the contribution notice is included.

DDJ

(Listings begin on page 24)

by Terje Bolstad

Terje Bolstad, Elektrokonsult AS, Konerudgt. 3, N-3000 Drammen, Norway.

BDOS and BIOS Calls for C

Listing One (Text begins on page 22)

```
/*******************************************************
*                                                      *
*  CP/M BDOS- AND BIOS- CALLS FOR C/80                 *
*                                                      *
*  Contributed by T. Bolstad, ELEKTROKONSULT AS        *
*  Konnerudgaten. 3, N-3000 Drammen, NORWAY.           *
*                                                      *
*  Date:  January 17, 1983.                            *
*                                                      *
*******************************************************/



/*  DEFINITION OF BDOS FUNCTIONS                       */

#define RESET     0    /*  SYSTEM RESET               */
#define CONSIN    1    /*  CONSOLE INPUT              */
#define CONSOUT   2    /*  CONSOLE OUTPUT             */
#define READIN    3    /*  READER INPUT               */
#define PUNOUT    4    /*  PUNCH OUTPUT               */
#define LISTOUT   5    /*  LIST OUTPUT                */
#define DIRCON    6    /*  DIRECT CONSOLE I/O         */
#define GETIOB    7    /*  GET I/O BYTE               */
#define SETIOB    8    /*  SET I/O BYTE               */
#define PRNTST    9    /*  PRINT STRING               */
#define READCB   10    /*  READ CONSOLE BUFFER        */
#define GETCST   11    /*  GET CONSOLE STATUS         */
#define RETVN    12    /*  RETURN VERSION NUMBER      */
#define RESDSK   13    /*  RESET DISK SYSTEM          */
#define SELDISK  14    /*  SELECT DISK                */
#define OPENF    15    /*  OPEN FILE                  */
#define CLOSEF   16    /*  CLOSE FILE                 */
#define SRCHFF   17    /*  SEARCH FOR FIRST           */
#define SRCHFN   18    /*  SEARCH FOR NEXT            */
#define DELF     19    /*  DELETE FILE                */
#define RDSEQ    20    /*  READ SEQUENTIAL            */
#define WRSEQ    21    /*  WRITE SEQUENTIAL           */
#define MAKEF    22    /*  MAKE FILE                  */
#define RENF     23    /*  RENAME FILE                */
#define RETLV    24    /*  RETURN LOGIN VECTOR        */
#define RETCD    25    /*  RETURN CURRENT DISK        */
#define STDMA    26    /*  SET DMA ADDRESS            */
#define GETAA    27    /*  GET ALLOCATION ADDRESS     */
#define WPDSK    28    /*  WRITE PROTECT DISK         */
#define GETROV   29    /*  GET READ/ONLY VECTOR       */
#define SETFAT   30    /*  SET FILE ATTRIBUTES        */
```

```
#define GETDPA  31    /*  GET DISK PARAMETERS ADDRESS  */
#define SGUC    32    /*  SET/GET USER CODE            */
#define RDRAN   33    /*  READ RANDOM                  */
#define WRRAN   34    /*  WRITE RANDOM                 */
#define COMFS   35    /*  COMPUTE FILE SIZE            */
#define SETRRC  36    /*  SET RANDOM RECORD            */
#define RESDRV  37    /*  RESET DRIVE                  */
#define WRRZF   38    /*  WRITE RANDOM WITH ZERO FILL  */

bdos(funct,arg)       /* corresponds to bdos((BC),(DE)) */
int     funct,arg;


/*  CALL EXAMPLE:    bdos(RETVN,0)
    BOTH ARGUMENTS MUST BE SPECIFIED !
    Values are returned IN HL.  BDOS errors
    are returned as -1.                          */


{
#asm
CPBASE  EQU     0         ;NORMAL 0-ORG'ED CP/M
CPNTRY  EQU     CPBASE+5  ;BDOS ENTRY

        POP     H         ;GET RETURN ADDRESS
        POP     D         ;GET ARG (INFORMATION ADDRESS)
        POP     B         ;GET FUNCTION NO.
        PUSH    B         ;RESTORE STACK
        PUSH    D
        PUSH    H

        PUSH    B         ;SAVE FUNCTION NO. ON STACK
        CALL    CPNTRY    ;BDOS CALL
        XCHG              ;SAVE HL IN DE
        MOV     L,A       ;SAVE A IN L
                          ;SIGN EXTENSION TO H:
        RLC               ; GET SIGN BIT INTO CY
        SBB     A         ; IF CY=0, RESULT AFTER SBB IS ZERO
                          ; IF CY=1, RESULT AFTER SBB IS -1 (IE ALL ONES)
        MOV     H,A       ; NOW A IS MOVED TO HL WITH SIGN EXTENSION
        POP     B         ;GET FUNCTION NO IN BC
        MOV     A,C       ;GET FUNCTION NO IN A

        CPI     12        ;WAS IT 'RETURN VERSION NUMBER' ?
        JZ      RETHL1
        CPI     24        ;RETURN LOGIN VECTOR ?
        JZ      RETHL1
        CPI     27        ;GET ALLOCATION ADDRESS ?
        JZ      RETHL1
        CPI     29        ;GET READ/ONLY VECTOR ?
        JZ      RETHL1
        CPI     31        ;GET DISK PARAMETER ADDRESS?
        JZ      RETHL1
        JMP     BDOSRET
```

(Continued on next page)

BDOS and BIOS Calls for C

Listing One (Listing continued, text begins on page 22)

```
RETHL1: XCHG
BDOSRET: RET                    ;WITH RETURNED VALUE IN HL

#endasm
}

/*      DEFINITION OF BIOS FUNCTIONS      */

#define BOOT      0    /*  COLD-BOOT                */
#define WBOOT     1    /*  WARM-BOOT                */
#define CONST     2    /*  CONSOLE STATUS           */
#define CONIN     3    /*  CONSOLE INPUT            */
#define CONOUT    4    /*  CONSOLE OUTPUT           */
#define LIST      5    /*  LIST DEVICE              */
#define PUNCH     6    /*  PUNCH                    */
#define READER    7    /*  READER                   */
#define HOME      8    /*  HOME DISK DRIVE HEAD     */
#define SELDSK    9    /*  SELECT DISK DRIVE        */
#define SETTRK   10    /*  SET TRACK                */
#define SETSEC   11    /*  SET SECTOR               */
#define SETDMA   12    /*  SET DMA ADDRESS          */
#define READ     13    /*  READ ONE SECTOR          */
#define WRITE    14    /*  WRITE ONE SECTOR         */
#define LISTST   15    /*  LIST STATUS              */
#define SECTRAN  16    /*  SECTOR TRANSLATION       */


bios(funct,arg1,arg2) /* corresponds to bios(function,(BC),(DE)) */
int funct,arg1,arg2;

/*  CALL EXAMPLE:     bios(SETTRK,5,0)
ALL 3 ARGUMENTS MUST BE SPECIFIED, even though
the last one is only used by SELDSK and SECTRAN. */

{
#asm

        POP     D       ;RETURN ADDRESS
        POP     H       ;ARGUMENT 2
        SHLD    ARG2S   ;SAVE IT
        POP     B       ;ARGUMENT 1
        XCHG            ;GET RETURN ADDRESS INTO HL

        POP     D       ;FUNCTION NO.

        PUSH    D       ;RESTORE SP
        PUSH    B
        PUSH    B
        PUSH    H       ;RESTORE RETURN ADDRESS

        PUSH    D       ;SAVE FUNCTION NO. ON STACK

        LXI     H,0     ;CALCULATE OFFSET ADDRESS FROM FUNCTION:
        DAD     D       ; GET FUNCTION NO. (OFFSET) IN HL
```

```
        DAD     H       ; 2*OFFSET
        DAD     D       ; 3*OFFSET
        XCHG            ; SAVE OFFSET ADDRESS IN DE

        LHLD    CPBASE+1 ;GET POINTER TO BIOS WBOOT ENTRY
        DCX     H       ;DECREMENT TO
        DCX     H       ;  POINT TO
        DCX     H       ;    START OF BIOS ENTRY JUMP TABLE

        DAD     D       ;ADD OFFSET (RESULT IN HL)
        XCHG            ;GET RESULT IN DE
        LXI     H,RET1
        PUSH    H       ;SAVE RETURN ADDRESS ON STACK

        LHLD    ARG2S   ;GET ARGUMENT 2
        XCHG            ;GET ARGUMENT 2 INTO DE
                        ; AND BIOS FUNCTION ENTRY ADDRESS INTO HL

        PCHL            ;GO TO BIOS

RET1:   XCHG            ;SAVE HL IN DE
        MOV     L,A

        RLC             ;GET SIGN BIT INTO CY
        SBB     A       ;IF CY=0, RESULT AFTER SUBB IS ZERO
                        ;IF CY=1, RESULT AFTER SUBB IS -1 (IE ALL ONES)
        MOV     H,A
        POP     B       ;GET BIOS FUNCTION NO. IN BC
        MOV     A,C
        CPI     9       ;SELECT DISK FUNCTION ?
        JZ      RETHL2
        CPI     16      ;SECTOR TRANSLATION FUNCTION ?
        JZ      RETHL2

        JMP     RETBIOS
RETHL2: XCHG            ;RETURN VALUE IN HL
RETBIOS: RET
ARG2S:  DS      2

#endasm
}
```

End Listing One

Listing Two

```
#include "cpmcall.c"
main()
{
        bios(CONOUT,bdos(RETCD,0)+'A',0);       /* print current disk */
}
```

End Listing Two

Printing Graphics Using the IBM PC

by Dan Daetwyler

The combination of the Color/Graphics Adapter and a printer capable of "dot graphics" provides a powerful utility function. The figure on page 36 shows an example of the image produced. PC-DOS at levels v.1.0 and v.1.1 does not provide direct support for this function. As this article is being written, v.2.0 was announced. I've heard that the new release corrects this omission, but my local suppliers can't supply v.2.0 as yet, so I'll ignore this new capability. All references to PC-DOS in this article should be assumed to refer to v.1.0 or v.1.1.

I've attempted to keep this at a "general" level, so that those of you who have "different" hardware may use the information to write your own utility. Specifically, however, the article (and included listing) is written for the standard IBM Color/Graphics Adapter and the original IBM Printer (EPSON MX-80) *with the GRAFTRAX PLUS ROM installed!* As I'm sure many of you are aware, the MX-80 produces quite variable results depending upon which control ROM is installed. The Graphics printer, provided on later models, should also work "as is," but minor adjustments may be required.

To introduce the "solution," it is first necessary to define the problem. In its simplest form, the problem is that of mapping. The color/graphics regeneration buffer contains a bit-encoded form of the displayed image. Dot matrix printers also use a bit-encoded form of the image to be printed. As you might expect, the two encoded forms are not even similar. The requirement for this utility, then, is simply that of mapping one bit-encoded form to another. Of course, before mapping can be performed, it is first necessary to understand the "rules" for encoding in both the input and output forms.

Dan Daetwyler, Route 5, Box 518A, Springdale, Arkansas 72764.

References are made to IBM-PC, PC-DOS, etc., which are registered trademarks of International Business Machines Corporation, Armonk, N.Y., and to EPSON MX-80, GRAFTRAX PLUS, etc., which are registered trademarks of Epson America, Inc.

Display Regeneration Buffer Format

For detailed specification of the input form, you are referred to the *IBM Technical Reference Manual* for the PC. This article considers only the medium-resolution graphic mode, and includes the basic information that describes the bit stream used in this mode. In this mode the addressable unit is the "pel," and the screen image is composed of 64,000 pels. The pels are "compressed" so that each byte of the buffer contains four pels, hence the regeneration buffer size is 16,000 bytes. Due to the compression, each pel must be two bits in size, giving four possible bit patterns (0 through 3). The zero pattern indicates the "background" color, while patterns 1 through 3 indicate which of the three possible "foreground" colors has been selected. Palette selection and background color selection are not germane to this article, so I'll ignore them.

So far, this encode seems simple. You need only visualize a string of bytes which are associated with the screen positions. The first byte of the buffer is the upper left corner of the screen. The four pels corresponding to the first four dots (left to right) at the upper left corner of the screen are in this byte. The second byte contains the next four dots, etc., until 320 dots have been defined (80 bytes). Unfortunately, this simple and straightforward representation now breaks down. The second row of dots on the screen is not the next 80 bytes of the buffer, but rather is the 80 bytes at an offset of 8000 in the buffer. The regeneration buffer is organized into two blocks or banks of 8000 bytes each. I won't bore you with the rationale behind this organization, since it's purely related to hardware design and addressing. From our viewpoint it's sufficient to know that even-numbered rows are stored in the first bank, while odd-numbered rows are in the second bank.

To summarize, medium-resolution graphic images appear in the regeneration buffer as they appear on the screen, left to right, top to bottom, but the pels are compressed four to a byte, and alternate rows are stored in alternate banks.

Print Line Buffer Format

Dot matrix printing, on the other hand, considers a print "line" to contain eight rows of dots, left to right, top to bottom. The most significant bit of each byte corresponds to the top row of dots in the print line. The printer used for the program provided offers two forms of dot matrix print: normal and high density. High density gives maximum resolution, so that was chosen. When operating in dot matrix, high-density mode, the printer will record 960 columns in the maximum print line (a column is a "dot"). Although one could map the graphic screen on a pel-to-dot basis, the image would be small, and no simulation of color would be possible. Since the maximum screen line contains 320 pels, dividing 960 by 320 gives us three dots per pel available. A little experimentation or computation shows that using a vertical dimension of two gives an image result that approximates screen proportion. Our mapping function, then, will map a pel to a 3x2 dot matrix. This also permits control of the dot density in this matrix to simulate color. Mapping for 100% density gives a very dark pel, while mapping for 30% density gives a light pel.

Now we can rough out the mapping function required. It must select pels from the regen buffer, by unpacking. These pels must be mapped to the corresponding 3x2 position in the printer buffer (left to right). The mapping function must also determine the row number of the pel and select from either the first bank of the buffer for even-numbered rows or the second bank for odd-numbered rows.

Program Description

Having determined the method of mapping, let's look at one program (see listing on page 38) that uses this scheme. PRTGRPH is a small, totally self-contained utility that will perform this mapping. It is written to produce a ".COM" file, since such a file is easier to patch if a change in control streams is required. The constants and data areas are given at the start of the program, but we'll skip over all of them initially.

The program proper starts with the control procedure, PRTGRPH. Initialization consists of setting the extra segment register to the segment address of the regen buffer (0B800H), and using standard DOS functions to prompt for (and accept) a title. The title is then "terminated" by a "$" and left in the title save area, LBUF, for later use. The printer is initialized by output of the control stream, LSPC. For the standard GRAF-

TRAX PLUS ROM, the printer control string:

Escape, A, 8

sets the printer to a line height of 8/72 inches. If you're using a different ROM, this control stream must be modified to match your ROM. A little experimentation using DEBUG to modify the control string should allow you to determine the proper line spacing for your printer.

Initialization is completed by setting the line count register (CX) to 50 (twice the lines on the screen), and the regen buffer register to a zero offset (SI). The main process loop at GRP1 is then entered. The call to PUTLNE will cause one print line to be generated and printed. The buffer register is stepped to the offset (start) of the next four display lines to be printed, and the process loops when the line count is zero. The remainder of the code in PRTGRPH simply restores the printer to normal spacing and mode, prints the saved title, and restores the printer page to the start of the next page.

The next procedure, PUTPRT, simply puts a string (terminated by a "$") to the printer. The code makes the function obvious, and I won't waste space describing this procedure.

PUTLINE clears the print buffer, builds the dot image, and then prints it. It's called once per line and is inserted in the program flow to preserve the entry values of the line counter and the regen buffer register. Useful work is performed by CLRBUF, BLDLNE, and PRTLNE which are called by this procedure.

CLRBUF again is a straightforward routine that simply pre-sets the printer buffer to binary zeros. PRTLNE simply prints the buffer, but in this case, it initializes the printer for dot matrix printing. This is done by output of the string BGRPH, which for the utilized ROM is:

Escape, L, 192, 3

This cryptic string tells the printer to enter high-density ("L") print mode for a line length of 192 + (3 * 256), or 960 dots. This re-initialization is required for each line, since the printer simply enters and stays in dot matrix mode for the byte count defined by this line length. It then returns to the mode it was in when it started the process.

BLDLNE is the basic mapping function. It begins by setting the print buffer register (DI) to zero, and an internal loop counter (BX) to the number of bytes in the line (80). A two-bit mask (0C0H) is set in the DH register, and the inner loop counter (CH is set in the outer loop, OLP. The inner loop, ILP, begins by clearing the shift count (CL) and loading the first regen byte for this print line into the AL register. This byte is masked by the content of DH and the result tested for zero. If zero, this pel is background color and is skipped. If non-zero, the call to SAVPEL is executed. In either case, the logic flow picks up at PEL2 which steps the shift count by 2 and loads the next regen data byte.

SAVPEL, which will be discussed in more detail later, simply fills in the 3x2 matrix with a dot pattern suitable for the color code determined by this masking process. Since the print line image is being built a pel at a time, left to right, the code at PEL2, PEL3, and PEL4 loads data from the appropriate bank of the regen buffer, and at the appropriate offset. Consider the first use of the routine. The SI register contains zero on entry, so the regen data load at ILP is of the first byte of the regen buffer, which corresponds to the first three dots in the first two rows of the print line (3x2 matrix again). The load at PEL2 is for the second two rows of the dot matrix print line, which correspond to the second row, first byte of the displayed image. It therefore is an odd-numbered row and must load from the second bank of the regen buffer (SI+2000H). The load at PEL3 is back to an even-numbered row, but it is now the third row of the displayed image (row 2 when considering zero as the first row), and must load from the first bank, second line, or offset 50H (80 characters into the buffer). The load at PEL4 follows the same logic, but it is again an odd-numbered row so the loading offset is 2050H.

Reaching STP, the code has completed the first three columns of the print line (eight rows), so the print buffer register is stepped by three. The mask is shifted to the next pel position, and the inner loop count decremented. This inner loop will execute four times (for the four pels in the screen byte), and then exit to a step of the regen buffer register, moving to the next four pels. The outer loop counter (BX is decremented, and the process continues until 80 regen buffer bytes have been processed. This corresponds to one display line, and the build function exits.

To complete the code description, SAVPEL simply converts the color code masked from the pel to a dot matrix code of suitable density and stores this code in the print line buffer. You'll note that the mask saved in the print line buffer is saved in a two-bit space (the two part of the 3x2 matrix) and is saved in three bytes (the three part). Since we don't know in which two-bit field the pel is located in this procedure, the test checks all four two-bit fields. It checks first for the presence of a "one bit" in the two-bit fields. If not found, it assumes that the color code is a two, since the routine would not be entered if the color code is zero. If a one bit is found, then a two bit is checked. If not found, the color code is one, while if found, the color code is three. The DL mask and the CL shift may be modified to give different shade patterns. The one provided in the code seems to give a good contrast between colors, yet provides a pleasing overall density. You can try other combinations and select the one most suited to your printer (and your eye).

That's it. The listing is relatively well commented, so try reading through the code if this description doesn't seem clear.

Assembly and Link

Key in the source code as shown in the listing, and assemble using either of the assemblers. If you have a different printer or control ROM, you should change the printer control strings to match your configuration. Link as you would any other program. The link will respond with a "warning" message informing you that there is no stack segment and will give an error count of one. Ignore this error message and execute EXE2BIN. Rename the resultant file "name.COM" and execute.

For those of you who are operating at PC-DOS v.1.0, you may not have the DOS utility EXE2BIN, which is almost necessary to produce a ".COM" file. You may, therefore, prefer an ".EXE" file. If so, you must add a stack segment, initialize your data segment register, initialize the stack for the eventual return to DOS, and change the PRTGRPH procedure to a "FAR."

Use of the Utility

Use of the utility is primitive, with one big "hooker." The utility will print the content of the color/graphics regeneration buffer, *at the instant of execution.* How you invoke this utility without disturbing this buffer is a function of how you're configured. I'm working with both the monochrome and the color/display adapters, so it's easy for me. I simply arrange my graphics program so that it does *not* clear the graphics screen on exit, and switches back to the monochrome display before exit. That leaves the graphics regen buffer containing the image I want, and I can enter the command to execute this utility on the monochrome display. If you have only the color/graphics adapter, you'll have to be considerably more complex in your process. One trick is to include code in your display generation program that writes the entire regen buffer to disk under some condition. This utility can then be simply modified to load the buffer image into memory and point to it instead of the regen buffer.

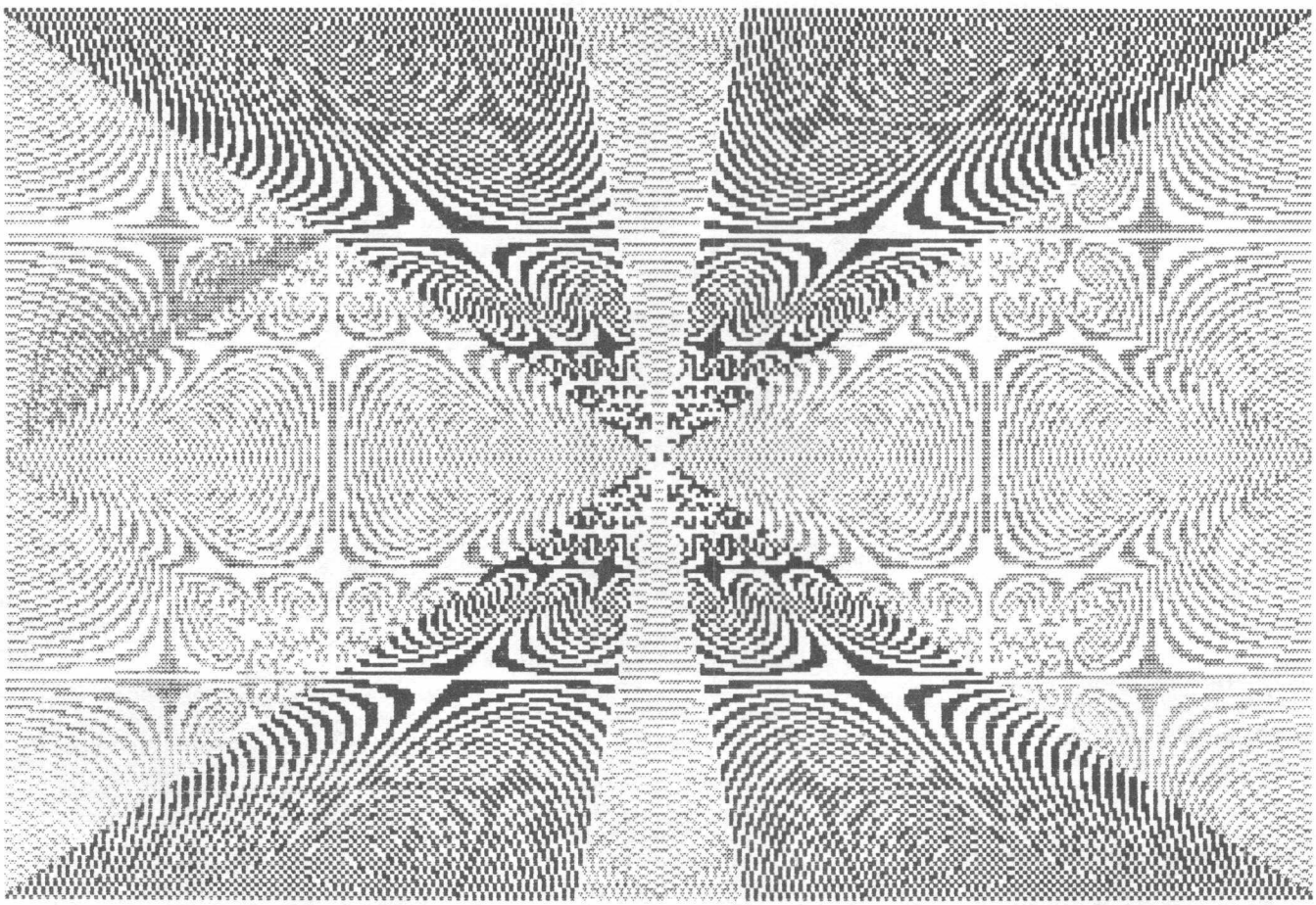
Finally, this program is hereby placed in public domain. You may copy it, utilize it in your own work, etc. freely and without notification.

Although this utility is small and not too much keying is required, I will provide copy service. A single-surface diskette and a check for $5 will get you a copy of the program source materials, object file, and an executable ".COM" file. A check for $7.50 will get you a new Verbatim (or equivalent) containing the same materials. I'll furnish the return mailer and will ship UPS Blue Label unless otherwise instructed. Send to: Dan Daetwyler, Route 5, Box 518A, Springdale, AR 72764 (that's Arkansas, not Arizona).

If you have trouble/questions with this one, you can write me at the above address, or call 501-756-0212. (I'm usually around 24 hours, seven days.) Good luck and happy printing.

DDJ

(Listing begins on page 38)



Kaliedoscope

Figure 1

Printing Graphics on the IBM PC (Text begins on page 32)

```
 COMMENT \
*****************************************************************************
*                  Copyright 1983 - DD Systems - Springdale, AR             *
*****************************************************************************
*                                                                           *
*       Output Color Graphic Screen to Printer                              *
*                                                                           *
*       This self-contained utility maps the "pel" maps of the color graphic *
*       refresh buffer to the printer.  It assumes the image has been       *
*       loaded by other services.                                           *
*                                                                           *
*****************************************************************************\

;
CODE    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CODE,DS:CODE
        ORG     100H
BEGIN:  JMP     PRTGRPH
;
PRMPT   DB      'Enter Title: $'        ;Prompt for title line
LSPC    DB      27,'A',8,13,10,'$'      ;Sets printer to 8/72" spacing
BGRPH   DB      27,'L',192,3,'$'        ;Sets to hi-density dot graphics
NORM    DB      27,'@','$'              ;Resets to normal mode
CRLF    DB      13,10,'$'
;
LBUF    DB      80
        DB      82 DUP (?)              ;Title buffer
PBUF    DB      960 DUP (?)             ;Bit graphics buffer
        DB      '$'

;
PRTGRPH PROC    NEAR
        MOV     AX,0B800H
        MOV     ES,AX                   ;Addressability of regen buffer
        MOV     DX,OFFSET PRMPT
        MOV     AH,9
        INT     21H                     ;Prompt for caption
        MOV     DX,OFFSET LBUF
        MOV     AH,10
        INT     21H                     ;Get title line
        MOV     AL,LBUF+1               ;Get entry length
        CBW
        MOV     SI,AX
        MOV     LBUF+2[SI],'$'          ;Set line terminator
        MOV     SI,OFFSET LSPC
        CALL    PUTPRT                  ;Set printer
        MOV     CX,50                   ;Print line count
        MOV     SI,0                    ;Start of regen buffer
GRP1:   CALL    PUTLNE                  ;Output one line
        ADD     SI,160                  ;  which is four screen rows
        LOOP    GRP1
        MOV     SI,OFFSET NORM
        CALL    PUTPRT                  ;Restore printer to normal
        MOV     SI,OFFSET CRLF
        CALL    PUTPRT                  ;Space line
        MOV     SI,OFFSET LBUF+2
        CALL    PUTPRT                  ;Output title
        MOV     CX,31
GRP2:   MOV     SI,OFFSET CRLF
        CALL    PUTPRT                  ;Space to end of page
        LOOP    GRP2
        MOV     SI,OFFSET NORM
        CALL    PUTPRT
        RET
```

```
PRTGRPH ENDP
;
;       Simple "put string" to printer
;
PUTPRT  PROC    NEAR
        PUSH    SI
        PUSH    DX
PPLP:   MOV     DL,BYTE PTR[SI]
        CMP     DL,'$'                  ;Check for "end of string" mark
        JZ      PPDNE
        MOV     AH,5
        INT     21H                     ;DOS Printer call
        INC     SI
        JMP     PPLP
PPDNE:  POP     DX
        POP     SI
        RET
PUTPRT  ENDP
;
;       Output one graphic "line"
;
PUTLNE  PROC    NEAR
        PUSH    SI
        PUSH    CX
        CALL    CLRBUF                  ;Clear bit buffer
        CALL    BLDLNE                  ;Build dot image
        CALL    PRTLNE                  ;Output line
        POP     CX
        POP     SI
        RET
PUTLNE  ENDP
;
;       Clear line buffer to null
;
CLRBUF  PROC    NEAR
        XOR     DI,DI
        MOV     AX,DI
        MOV     CX,480
CBLP:   MOV     WORD PTR PBUF[DI],AX    ;Clears line buffer to null
        INC     DI
        INC     DI
        LOOP    CBLP
        RET
CLRBUF  ENDP
;
PRTLNE  PROC    NEAR
        MOV     SI,OFFSET BGRPH
        CALL    PUTPRT                  ;Set printer for dot graphic line
        MOV     SI,OFFSET PBUF
        MOV     CX,960                  ;  and output bit stream
PL1:    MOV     DL,BYTE PTR [SI]
        MOV     AH,5
        INT     21H
        INC     SI
        LOOP    PL1
        RET
PRTLNE  ENDP
;
;       Build dot graphic bit stream in buffer
;
BLDLNE  PROC    NEAR
        XOR     DI,DI                   ;Dot buffer index
        MOV     BX,80                   ;Loop count for line
OLP:    MOV     DH,0C0H                 ;Outer loop mask
        MOV     CH,4                    ;Inner loop counter
ILP:    XOR     CL,CL                   ;Shift for pel save
```

```
        MOV     AL,BYTE PTR ES:[SI]     ;Get screen byte
        AND     AL,DH                   ;Mask
        JZ      PEL2
        CALL    SAVPEL                  ;non-zero, so print blob
PEL2:   ADD     CL,2
        MOV     AL,BYTE PTR ES:[SI+2000H] ;Next row
        AND     AL,DH
        JZ      PEL3
        CALL    SAVPEL
PEL3:   ADD     CL,2                    ;Stepping shift count
        MOV     AL,BYTE PTR ES:[SI+50H] ;Row three
        AND     AL,DH
        JZ      PEL4
        CALL    SAVPEL
PEL4:   ADD     CL,2
        MOV     AL,BYTE PTR ES:[SI+2050H] ;Fourth row
        AND     AL,DH
        JZ      STP
        CALL    SAVPEL
STP:    ADD     DI,3                    ;Step save buffer pointer
        SHR     DH,1
        SHR     DH,1                    ;Reposition pel mask
        DEC     CH
        JNZ     ILP                     ;Continue inner loop for next col
        INC     SI                      ;Step regen buffer ptr
        DEC     BX
        JNZ     OLP                     ;Outer loop
        RET
BLDLNE  ENDP
;
;       Save bit image for this pel - different "shade" for each color
;
SAVPEL  PROC    NEAR
        TEST    AL,55H                  ;One bit on
        JNZ     ONEB
        JMP     DOTWO
ONEB:   TEST    AL,0AAH                 ;Test for two bit
        JNZ     DOTHRE
;   The following is for a data bit of 01
        MOV     DL,0C0H                 ;Load mask
        SHR     DL,CL                   ;  and position
        OR      PBUF[DI],DL
        OR      PBUF[DI+1],DL
        OR      PBUF[DI+2],DL
        RET
DOTWO:  MOV     DL,80H
        SHR     DL,CL
        OR      PBUF[DI],DL
        OR      PBUF[DI+2],DL
        SHR     DL,1
        OR      PBUF[DI+1],DL
        RET
DOTHRE: MOV     DL,0C0H
        SHR     DL,CL
        OR      PBUF[DI+2],DL
        RET
SAVPEL  ENDP
;
CODE    ENDS
;
        END     BEGIN
```

End Listing

The Game of Life
on the IBM-PC

by Simson L. Garfinkel

Computer programs for running John Conway's Game of Life were quite popular a few years ago. The problem with these older simulations was usually their speed: a single generation on a ten-by-ten grid could take up to 90 seconds. Presented here is a version of the game for the IBM/PC computer. Features of this version are adjustable speed (with 2.7 generations/sec as a top speed), easy entry of the seed generation (via a screen-editor), and marking of cells due to be "born," and cells remaining alive each generation (as different from cells marked to die). This feature gives a better impression of fluidity on the grid.

This version requires only the IBM monochrome display to run, and will run under either MS-DOS or CP/M-86, since only calls to the IBM ROM are made.

Background

The Game of Life takes place on a square grid. Every cell on the grid can be either alive or dead. Every cell on the grid has eight neighbors. Each generation, every cell on the grid is evaluated. The cell will remain alive if it has two or three neighbors, or will die if it has less than two (from loneliness), or more than three (from overpopulation). An empty cell will have a "birth" (a live cell will be placed there the next generation) if it has exactly three neighbors. The game was developed by John Conway in 1976.

The Program

The program is straightforward. When first run, the screen is cleared and the cursor is positioned in the center. The cursor may be moved by pressing the four arrow keys on the keyboard (Num Lock doesn't matter because the keyscan code is interrogated, not the ASCII code). A live cell may be deposited by depressing Ins, and a cell may be cleared by pressing Del. When the screen is complete, pressing Esc starts evaluation of the next generation.

The program stores the grid in the screen memory of the monochrome display. Associated with every displayable character on the screen are two bytes of memory, the low byte for the character displayed, and the hi byte for the display attributes (underlined, reversed, etc.). The program uses the attribute byte of every screen position as a second array to hold the next generation in while it evaluates the present generation.

After the time delay, subroutine *count* is called. *Count* counts the number of neighbors that every cell has, and decides whether or not the cell will be alive in the next generation. If it is going to be alive, the display attribute for that character is set to *rev*. If the cell is going to be dead, the attribute is set to *dark*. *Rev* and *dark* are reverse video and normal video in my listing, so when a cell is going to have a birth or stay alive, it is inverted on the screen, but *rev* and *dark* can be changed to three and five, causing both to display as normal if the flickering this produces is annoying.

After all of the decisions regarding life and death are made, subroutine *update* goes through screen memory putting in the character for a live cell if the cell is supposed to be alive, or a dead cell if the cell is supposed to be dead, and resetting the attribute byte.

The program then looks for a user key press. If one has occurred, the program quits if it was an Esc, or changes the time delay value if it was a digit. Pressing 0 gives no delay, or about 2.7 generations per second. Pressing 9 gives a 3.5 second delay per generation.

Expansions

The one problem with this implementation is that the screen doesn't have enough rows to allow a simulation of a complex colony. I would love to see an implementation of life in medium-resolution color graphics, with births in blue and deaths in red, but I don't have access to a color display, so that will have to wait. The program could also be cleaned up to make it run faster, but 2.7 generations a second is really fast enough for most applications. DDJ

(Listing begins at right)

Simson L. Garfinkel, 18 Dartmouth Lane, Haverford, Pennsylvania 19041.

Game of Life (Text begins on page 42)

```
Comment $

        ***********************************
        *                                 *
        *        The Game of Life         *
        *                                 *
        ***********************************

        John Conway's mathematical game of life, implemented on
        the IBM/PC, by Simson L. Garfinkel.

        Written in 8088 assembly language using the Microsoft
        Macro Assembler.

        Notes on running the program:

          When program is run:

                1.  Screen clears.
                2.  User enters first generation from keyboard.
                    Arrow keys move the cursor.  INS key deposits
                    a live cell, DEL removes a live cell, (in case
                    the user makes a mistake.)
                3.  Pressing ESC starts program.
                4.  For each generation, cells which will have life
                    on the next turn are inverted.
                5.  Screen is updated to next generation.
                6.  Keyboard is interrogated for command.
                7.  If ESC is pressed, program terminates.
                8.  If a number 0-9 is pressed, speed is selected.
                    At speed 0, approx. 2.7 generations/sec are performed
                    At speed 9, each generation takes 3.5 sec.
                9.  program loops to #4.

        $

        ;global definitions

live    equ  02          ;character for a live cell
dead    equ  00          ;character for a dead cell

rev     equ  70h         ;reverse video (marks cell to live)
dark    equ   2          ;normal video  (marks cell to die )

time    equ  300         ;time delay base

TERM    equ  27          ;Character to exit mode


cseg    segment para public 'code1'
start   proc far
        assume cs:cseg,ds:nothing,ss:stack,es:nothing

                         ;set up return location
```

(Continued on next page)

Game of Life (Listing continued, text begins on page 42)

```
        push ds
        sub  ax,ax
        push ax             ;now I can go home when I'm finished.


        call Enter          ;Enter board
        mov  cx,0           ;initial delay, 0
main:
        push cx             ;save delay variable
        cmp  cx,0
        jz   s13

s1:     push cx
          mov  cx,time
s11:      push cx
            mov  cx,time
s12:        loop s12
          pop  cx
          loop s11
        pop  cx
        loop s1             ;what a time delay!

s13:    call count          ;Count up every cell's neighbours,
        call update         ;Update screen
        pop  cx             ;get back the time delay

        mov  ah,1           ;See if user has pushed a key
        int  16h
        jz   main           ;nope - loop back

        mov  ah,0           ;get the character our of the buffer
        int  16h

        cmp  al,TERM
        jnz  s2
        ret                 ;finished - go back to ms/dos

s2:     cmp  al,'0'         ;see if it is a speed command
        jb   main
        cmp  al,'9'
        jnbe main
                            ;It's a number
        sub  al,'0'         ;now it goes from 0 to 9
        mov  ah,0
        mov  cx,ax          ;put it in cx
        jmp  main
start   endp



Enter   proc near           ;Subroutine to enter board
                            ;define scan-codes:
left    equ  75
right   equ  77
up      equ  72
```

```
down    equ  80
point   equ  82
del     equ  83
esc     equ   1

        call cls          ;clear the screen

        ;Registers are used as follows:
        ;DH - Y position
        ;DL - X position

        mov  dh,12
        mov  dl,40

e1:     mov  bh,0         ;move the cursor to x,y position
        mov  ah,2         ;code for cursor move
        int  10h          ;interrupt for cursor move

        mov  ah,0         ;set up to read the next keypress
        int  16h          ;keypress read

        cmp  ah,left      ;make a rational decision about the users
        jz   go_left      ;entry.
        cmp  ah,right
        jz   go_right
        cmp  ah,up
        jz   go_up
        cmp  ah,down
        jz   go_down
        cmp  ah,point
        jz   go_point
        cmp  ah,del
        jz   go_del

        cmp  ah,esc
        jnz  e1           ;loop back - unknown command
        mov  dx,23*256    ;put the cursor at lower-left hand corner
        mov  ah,2
        int  10h
        ret               ;go back to caller

go_left:                  ;move left if I can
        cmp  dl,0         ;in leftmost collumn?
        jz   e1           ;yes - go back
        sub  dl,1         ;no  - subtract one
        jmp  e1           ;go back

go_right:                 ;move right if I can.
        cmp  dl,79
        jz   e1
        add  dl,1
        jmp  e1

go_up:                    ;go up if I can
        cmp  dh,0
        jz   e1
        sub  dh,1
        jmp  e1
```

(Continued on next page)

Game of Life (Listing continued, text begins on page 42)

```
go_down:                    ;go down if I can
        cmp   dh,24
        jz    e1
        add   dh,1
        jmp   e1

go_point:                   ;put a live dot where the cursor is -- don't move it
        mov   al,live       ;it's the live character
gp2:    mov   cx,1          ;one character to write
        mov   ah,10         ;code to write character
        int   10h           ;do it
        jmp   e1            ;get next command

go_del:                     ;delete character at cursor
        mov   al,dead
        jmp   gp2           ;let go_point do the rest
Enter   endp



Cls     proc near           ;Subroutine to clear the screen
        mov   ax,6*256
        mov   cx,0
        mov   dx,24*256+79
        mov   bh,2
        int   10h
        ret
cls     endp


Count   proc near           ;Subroutine to count up every cell's neighbours
        ;Registers used:
        ;DH,DL:  Y,X of current cell being interrogated
        ;DS    :  Base offset - into screen memory
        ;DI    :  offset for character presently being looked at
        ;
        ;Outline for each character
        ;  1.  Count up number of neighbours
        ;  2.  If three neighbours, or if two and cell is live, put
        ;      a rev  on the screen at the attribute position, else
        ;      put a dark
        ;  3.  Go to next character


chk     macro  yy,xx
        local  ch1,offs
offs    equ    (xx+yy*80)*2
        mov    cx,[di+offs]          ;get byte to check
        cmp    cl,live               ;check to see if this cell is alive
        jnz    ch1                   ;nope
        add    al,1                  ;yes - increase neighbour count
ch1:
        endm
```

(Continued on page 50)

```
        mov   ax,0B000H
        mov   ds,ax         ;offset value for monochrome display

        mov   dh,1          ;Start at 1,1 and go to 23,78
        mov   dl,1          ;to prevent wrap-arround

c1:     mov   ax,160        ;get true offset from ds into screen memory
        mul   dh

        mov   cx,dx
        mov   ch,00         ;just get dl
        add   ax,cx
        add   ax,cx         ;ax:=(dh*80+dl)*2

        mov   di,ax         ;di:=ax
        mov   ax,0          ;ax will be used for neighbour counting

        chk   -1,-1         ;count number of neighbours
        chk   -1, 0
        chk   -1,+1
        chk    0,-1
        chk    0,+1
        chk   +1,-1
        chk   +1, 0
        chk   +1,+1         ;test all of the neighbours

        mov   cx,[di]       ;get byte to check
        cmp   al,3
        jz    give_life     ;life if has 3 neighbours
        cmp   cl,live       ;is it alive?
        jnz   give_death    ;no
        cmp   al,2          ;he lives if he has 2 neighbours and he is already
                            ;alive
        jnz   give_death    ;nope

give_life:                          ;make this one alive
        mov   ch,rev
        jmp   c2

give_death:
        mov   ch,dark
c2:     mov   [di],cx               ;put back on the screen

next_cell:
        cmp   dl,78         ;am I at the end of the X line?
        jz    c3            ;yes
        add   dl,1          ;nope
        jmp   c1
c3:     mov   dl,1
        cmp   dh,23         ;am I at the end of the Y line?
        jz    c4            ;yes
        add   dh,1          ;nope
        jmp   c1
c4:     ret                 ;yes - go home!
Count   Endp
```

(Continued on next page)

Game of Life (Listing continued, text begins on page 42)

```
Update  proc near               ;This updates the generation on the screen
        mov   ax,0B000H         ;Get screen offset
        mov   ds,ax

        mov   bx,24*80*2-2      ;loop through all of the screen but last line
u1:     mov   cx,[bx]           ;line
        cmp   ch,rev            ;is it to live?
        jnz   u2                ;no
        mov   cl,live           ;yes
        jmp   u3
u2:     mov   cl,dead           ;no
u3:     mov   ch,dark           ;turn off reverse
        mov   [bx],cx           ;put it back on the screen
        sub   bx,2              ;loop back until done with the screen
        jg    u1
        ret                     ;go back to caller
Update  endp
cseg    ends


stack   segment para stack 'stack'
        db 30 dup('stack   ')
stack   ends


        end
```

End Listing

Series Expansion in Forth

by Wendall C. Gates

Series expansions are widely used in computers to generate precision non-linear functions. The usual approaches to series implementation use a lot of constants. This results in much space being used for headers or for stuffing all the constants into arrays, which makes checking and adding more constants awkward. Also, the mechanics of computing each series varies – some have only even and others only odd exponents, some have all exponents, some begin with a constant and others do not, etc.

A different approach, which places coefficient-exponent pairs into an array while preserving readability and access, is shown in screens 38-43, at right. The algorithm is identical for all series, regardless of first-term constant or exponent pattern. This approach, like Dr. Eaker's Case statement, permits the programmer to concentrate on the results rather than on the mechanics.

The first goal is to compactly store the data, which consists of floating-point coefficients and integer exponents. SERIES-CONSTANTS expects on the stack the number of terms to be computed. It then allots space for this number (1 byte) plus 7 bytes per term – one byte for the integer exponent and 6 bytes for the floating-point coefficient. This arrangement limits the number of terms and the exponent to 256, which would seem adequate for most needs. A header is compiled for the assigned NAME and the number of terms is stored in the first byte of the allotted space.

Next, the coefficients and exponents are written in the eminently readable format shown in Screen 227 (on page 55) which puts them on the stack in the reverse of the order shown, with the smallest exponent on top. Screen 228 contains the code to obtain the proper sign and to reduce the angle to within less than 360°.

These constants and exponents are next loaded into the allotted space with the phrase "n NAME LOAD-CONSTANTS." The number of constants to be stored is checked against the number for which space was allotted, and the

Wendall C. Gates, PE, Advanced Instrumentation Inc., Box 2070, Santa Cruz, California 95063.

Series Expansion

```
  Screen # 38
 0 ( Series Expansion Statement by W.C. Gates   11-21-82 )
 1 0 VAL NEXTADD     F0 FVAL FACCUM
 2 0 VARIABLE POWER   0 VARIABLE NEXTEXP
 3 : 0VARS F0 TO FACCUM  0 POWER ! ;  ( Zero FACCUM, POWER  )
 4
 5 ( NGET does  z add ... z add n  with add+1 in NEXTADD )
 6 : NGET  DUP 1+ TO NEXTADD C@ ;
 7
 8 ( n SERIES-CONSTANTS  name    defines the storage space for an)
 9 ( array which contains [number of terms] [exp. of 1st term]   )
10 ( [FP coefficient of 1s term] [exp of 2nd term][next coeff.]  )
11 ( exponents rise monotonically, but gaps OK   )
12 : SERIES-CONSTANTS
13   <BUILDS DUP HERE C!    7 * 1+ ALLOT  ( for 6-byte floating- )
14    DOES> ;                             ( point numbers        )
15 -->


  Screen # 39
 0 ( Series Expansion-- LOAD-CONSTANTS )
 1 (                                                             )
 2 ( LOAD-CONSTANTS stores the pre-computed coefficients into the )
 3 ( space alloted by SERIES-CONSTANTS. These coefficients must be)
 4 ( on the stack already, together with the exponent for each    )
 5 ( Cn exp[n] C1 exp1 C0 exp0  n  add ... for example)
 6 : LOAD-CONSTANTS DUP 1+ TO NEXTADD   ( store add of 2nd byte  )
 7   >R DUP R>                          ( DUP n under address    )
 8   C@ ?PAIRS          ( fetch allotted n, QUIT if not same     )
 9   0 DO                               ( DO n times             )
10     NEXTADD C!                    ( store exponent at next byte )
11     NEXTADD 1+ TO NEXTADD         ( point to next byte          )
12     NEXTADD F!                    ( store coefficient           )
13     NEXTADD 6 + TO NEXTADD        ( point past stored coeff.    )
14     LOOP  ;
15 -->


  Screen # 40
 0 ( Series Expansion--continued  )
 1 ( TERM raises z to the power stated in the first byte of each )
 2 ( term/constant location, starting with the value z^n of the  )
 3 ( previous term.                                              )
 4
 5 ( z  z^n ... z  z^[n+x] z^[n+x] ,   NEXTADD points to exp.    )
 6
 7 : TERM
 8    NEXTADD C@ NEXTEXP !          ( store target exponent       )
 9     BEGIN
10      POWER @  NEXTEXP @  = 0=   ( current exp. = target exp?  )
11     WHILE                       ( if not,                     )
12      FOVER F* 1 POWER +!        ( raise exponent 1, inc. POWER )
13     REPEAT                      ( repeat until current=target, )
14   FDUP  ;                       ( then FDUP and exit.         )
15 -->
OK
```

```
  Screen # 41
 0 ( Series Expansion -- ADD-TO   )
 1
 2 ( ADD-TO fetches the next coefficient and multiplies it with )
 3 ( z^n on TOS, then adds result to FACCUM, and points        )
 4 ( NEXTADD to exponent of next term.                          )
 5
 6 ( z  z^n  z^n .... z  z^n  )
 7
 8 : ADD-TO
 9   NEXTADD 1+ TO NEXTADD        ( point to coefficent          )
10   NEXTADD F@ F*                ( fetch and multiply x z^n     )
11   FACCUM F+ TO FACCUM          ( add to cumulative total      )
12   06 NEXTADD + TO NEXTADD ;    ( point to exp. of next term   )
13
14 -->
15
```

```
  Screen # 42
 0 ( Series Expansion-- SERIES    )
 1 ( Used as  z [name] SERIES ... for value z.                 )
 2 ( number of terms is defined in [name]                      )
 3
 4 : SERIES
 5   OVARS  >R F1 R>      ( set variables to 0, F1 to stack )
 6   NGET 0 DO              ( get # of terms for loop       )
 7            TERM  ADD-TO  ( calculate term and add to total)
 8            LOOP
 9   FDROP FDROP            ( drop calculation values       )
10   FACCUM  ;              ( leave accumulated value on stack)
11   ;S
12
13
14
15
```

```
  Screen # 43
 0 ( SERIES expansion-- BY-N and BY-EXP  , fetching/storing exp.  )
 1 ( BY-N leaves the address of the nth exponent-coefficient pair )
 2 ( used as ....  n  NAME-CF  BY-N   followed by F@ or F!        )
 3
 4 : BY-N  2DUP @ >                    ( test for n inside array )
 5         IF CR ." n is outside array." QUIT ENDIF
 6         2 + SWAP 7 * + ;           ( point to coeff. in nth term)
 7
 8 ( .. n NAME-CF BY-EXP leaves the address of coeff. with exp. n )
 9
10 : BY-EXP NGET SWAP 0 FLAG !         ( n to TOS, add+1 to NEXTADD)
11          BEGIN DUP NEXTADD C@ =          ( compare to term exp  )
12          IF 1 ELSE OVER       FLAG @ =  ( no more than n terms )
13          IF CR ." Exponent not found." QUIT ENDIF
14          0 1 FLAG +! NEXTADD 7 + TO NEXTADD ENDIF ( next term  )
15          UNTIL 2DROP NEXTADD 1+ ;       ;S

OK
```

```
  Screen # 45
 0 ( Derivative Calculations--N derivatives of n-order polynomial.)
 1 ( Method from Hoffman, DDJ April 81, implm by W.C.Gates  Dec 82)
 2   F0 FVARIABLE EVAL-POINT  0 VARIABLE POLY-ARRAY
 3 : GEN.ARRAY
```

(Continued on next page)

exponent-coefficient pairs are stored into the space. Only one header is required for all the exponents and coefficients.

Computation is done by SERIES, which expects a floating-point value and the beginning address of the data space, left by NAME-CF of the series. The first byte is the number of terms, which is used as the DO-LOOP index. TERM raises the power of the base value until it matches the next stored exponent. ADD-TO fetches the matching coefficient, multiplies it with the raised base value, and accumulates the total in ACCUM. This sequence is repeated *n* times to generate and add up *n* terms. Exponents may increase in any monotonic pattern. A series in half-power exponents, frequently encountered in flow measurement, may be generated by taking the square root of the base value before using it to compute the series.

In some applications, adaptive control for example, it may be necessary to modify the series coefficients during operation. This feature is provided by BY-N, used as "n NAME-CF BY-N," which leaves the address of the *n*th coefficient in the series. The word BY-EXP similarly leaves the address of the coefficient of the term whose exponent is *n*.

Where evaluation of polynomial derivatives is required, a slower and less compact method is implemented, in Screens 45-49 (pages 53-55). The polynomial coefficients are transferred to a working array, with floating-point 0s inserted for the absent coefficients. This array is then processed by CALC.DERIVS, resulting in the coefficient values being replaced by the *n* derivative values, including the value of the polynomial itself in the first location. The values of the *n* derivatives are then fetched as required using N DERIV. For an explanation on the mathematical method, see the article by R.A. Hoffman in *Dr. Dobb's Journal,* April 1981.

The screens provided are written for a floating-point package which uses a 6-byte format for floating-point numbers. For 4-byte formats, change the 6s and 7s to 4 and 5. The floating-point coefficients are handled as groups of bytes, so the program is indifferent to the internal sign-exponent-mantissa format.

DDJ

(Listing begins on page 52)

```
 4   10 DUP                          ( highest anticipated order      )
 5   HERE DUP POLY-ARRAY !             ( beg. address to POLY-ARRAY   )
 6   C!                              ( store highest order in 1st byte )
 7   1+ 7 * 1+ ALLOT ;               ( allot space for order+1 terms,
 8                                 ( each of which is exp-byte plus float# )
 9
10 : GET.ORDER      ( expects add. of coef. array, fetches order,  )
11   DUP C@             ( which is exponent of last term, to TOS   )
12   1- 7 * 1+ +
13   C@ ;
14 GEN.ARRAY                       ( allots array when LOADed.        )
15 -->


  Screen # 46
 0 ( Derivative Calculations -- FILL.ARRAY transfers coeff's from )
 1 ( permanent array to working array, filling F0 for absent coefs)
 2 : FILL.ARRAY              ( poly-array-add  coef-array-add ....  )
 3   DUP GET.ORDER            (               padd  cadd  n  ....   )
 4   >R OVER R> DUP ROT C!              ( store order in 1st byte    )
 5   >R 1+ SWAP 1+ R> 1+ 0     ( point to exp.bytes, loop n+1 times)
 6   DO OVER C@ I =            ( print exp, test for equal exp's    )
 7    IF   DUP I SWAP C!                            ( if =, move exp. )
 8     1+ DUP 6 + SWAP ROT 1+ DUP 6 +       ( and calc. next add's )
 9     ROT ROT  SWAP >R
10     F@ R> F!  SWAP
11     ELSE DUP 7 + SWAP                          ( if not = stuff F0)
12     DUP I SWAP C!                        ( point to next work.add)
13     1+ >R F0 R> F!          ( but try again at same loc in source)
14     THEN                    ( array.                              )
15    LOOP 2DROP ;    -->


  Screen # 47
 0 ( Derivative Calculation -- *FACS multiplies terms by n!     )
 1
 2 : *FACS                             ( empty stack .... same )
 3   F1 POLY-ARRAY @ DUP 16 + SWAP ( F1>stack, point to ^2 term)
 4   C@ 1+ 2                           ( n+1, 2 = for 2 to n       )
 5    DO
 6     I SWAP DUP DUP >R >R >R      ( I>stk, add's to ret. stk   )
 7     S->D D>F                     ( convert I to floating #    )
 8     F* FDUP R> F@ F*             ( x prev. "!", x coeff,      )
 9     R> F!                        ( store product back         )
10     R> 7 +                       ( point to next coeff.       )
11     LOOP
12   DROP FDROP ;                   ( leave stack clean          )
13
14 -->
15

OK


  Screen # 48
 0 ( Derivatives Calculation -- CALC.DERIVS performs calculations )
 1 ( replacing coef's with values in working array.               )
 2 : CALC.DERIVS       ( <point of evaluation--F#> <series-name>..)
 3   >R EVAL-POINT F!             ( store X1 value to EVAL.POINT   )
 4   POLY-ARRAY @ DUP R>  ( work-add to stack, R> series-array-add)
 5   FILL.ARRAY C@ DUP     (   coefs into work-array, filling F0's)
 6   1- SWAP 0                ( ...N-1 N  0 )
 7    DO DUP DUP I 1- SWAP
```

Series Expansion

(*Listing continued, text begins on page 52*)

```
 8     DO                          ( limit is 1st index, index is  )
 9      I 1+ POLY-ARRAY @ BY-N F@     (  N-1 minus iteration #     )
10      EVAL-POINT F@ F*              ( prev.coeff. x point value )
11      I POLY-ARRAY @ BY-N DUP >R    ( add above to current coeff)
12      F@ F+ R> F!
13      1- -1 +LOOP DROP
14     LOOP
15     *FACS ;      -->                    ( each coef x n factorial)
   Screen # 49
 0 ( Derivatives Calculation-- DERIV fetches nth derivative value )
 1 ( checks for out-of bounds request, non-destructively )
 2 : DERIV
 3   DUP POLY-ARRAY @ DUP
 4   ROT SWAP C@ >
 5    IF 5 SPACES ." Requested too high a derivative.  " 2DROP
 6    ELSE 2+ SWAP 7 * + F@
 7    THEN ;   ;S
 8
 9            USE OF DERIVATIVES CALCULATION
10  Define largest order polynomial to be used; add 1 to order and
11  insert this value into Screen 45, line 4. Then 45 LOAD.
12
13  <F#evaluation point> <series-coeff-name> CALC.DERIVS... does
14  the calculations,  N  DERIV puts the nth derivative value on
15  the stack.

OK
   Screen # 227
 0 ( Floating Point-- SINE  )
 1 6 SERIES-CONSTANTS SINE-CF             ( allot storage space )
 2         ( coefficients              exponents  )
 3
 4 -2.50521084 X -8                      11       ( monotonically  )
 5  2.75573192 X -6                       9       ( decreasing     )
 6 -1.98412698 X -4                       7       ( exponents      )
 7  8.33333333 X -3                       5
 8 -1.66666667 X -1                       3
 9  F1                                    1
10 6 SINE-CF LOAD-CONSTANTS             ( n  [name] LOAD-CONSTANTS )
11 : [SINE] SINE-CF SERIES ;
12
13 ( "X" is used for floating point entry to permit using "E" in)
14 ( reading hex keypads. F1 is 1.0 floating point.             )
15 -->
   Screen # 228
 0 ( Trig Package -- SINE continued )
 1
 2 : ANGLE-PREP                      ( convert to rads if rqd      )
 3   FPI F/MOD FABS FIX D->S         ( get # of half circles       )
 4   DUP 0= IF DROP ELSE >R          ( x -1, is - for 2nd half     )
 5    F1        R> 0 DO              ( +1 for 1st, -1 for 2nd      )
 6    F1 FMINUS F* LOOP              ( apply sign to angle <180 d. )
 7    F* ENDIF ;                     ( calculate sine series       )
 8
 9 : FSIN
10   DEG-RAD 0= IF DEG>RAD ENDIF    ( using degrees? convert >rads)
11   ANGLE-PREP [SINE] ;            ( strip full turns, calc. and )
12                                  ( apply sign, calc. value     )
13 ;S
14
15
```

End Listing

Fast Matrix Operations
In Forth, Part 1

by Steven A. Ruzinsky

Steven A. Ruzinsky, 2110 S. Austin Blvd., Cicero, Illinois 60650.

It is fairly well-known that among various languages having software-implemented floating point, there is little practical difference in the speed at which number-intensive programs run. The reason for this small difference is that a large proportion of total program run time is spent on the floating-point routines that are written in assembly language and are therefore essentially the same for every language. Since most languages developed under this condition, it is doubtful that any need for faster numerical operations influenced their evolution. With the advent of hardware-implemented floating point in the form of numeric coprocessors such as the 8087, 80287, and 16081, a situation arises in which most of the popular numerical languages are incapable of efficiently utilizing their speed. This deficiency is most true of interpreted languages such as BASIC and APL.

In my quest for a faster alternative, I have found what I believe to be an ideal in an unpopular floating-point language, Forth. More specifically, I have found Polyforth for the IBM PC to be best for floating-point applications because it uses the 8087 register stack as an extra Forth stack and it includes the 8087 instruction set in its assembler. By using the assembler, it is not unusual for a numerically intensive program to run 100 times faster in Forth than in standard BASIC.

From the viewpoint of programming convenience, APL is the Rolls Royce of numeric languages. The enthusiasm of APL users easily matches that of Forth users, although each group loves their language for entirely different reasons. APL is characterized by syntactically simple commands that operate on entire arrays rather than on single numbers. Generally, the shortest program for performing a given task can be written in APL. APL does, however, have a reputation for being slow.

I have found that it is not at all difficult to write Forth programs that resemble APL commands. For example, consider the following phrase for performing matrix multiplication in Forth:

```
' A ' B ' C A*
```

Here, A* is the matrix multiplication operator and A, B, and C are arbitrary conformable matrices. In algebraic notation, C = AB. The advantage of Forth over APL is, of course, speed. The program represented above will multiply two 10th-order square matrices in 0.1 seconds. APL users are invited to present benchmarks for comparison.

In this first of a three-part series, some fundamental Forth matrix utilities will be presented. In the second installment, programs for the classical matrix operations such as matrix addition, multiplication, and inversion, plus some equally useful but more unusual operations, will be presented. The third part will be concerned with matrix applications. If reader response is favorable, more items on matrix applications could be forthcoming.

These programs require: (1) the IBM PC with more than 64 KB, preferably 320 KB, of RAM; (2) an 8087 numeric coprocessor installed in the empty socket next to the 8088 (dipswitch SW1-2 must be off); and (3) Polyforth Level 2 for the IBM PC from Forth, Inc.

The Programs

The fundamental programs are presented in screens 207 to 209 (page 57). A description of these programs follows:

a!, a@, etc. – Fast concatenated 16-bit variable stores and fetches. These will be used within many of the succeeding programs.

matrix – Defines a character string as a matrix. Usage example:

```
10 20 matrix X
```

Here, X is defined as a matrix of 10 rows by 20 columns. Such a matrix has the following properties:

- The data contained in the matrix resides outside the Forth dictionary and can occupy all available memory beyond the first 64 KB segment of RAM.
- Two 64-bit numbers can be stored in each matrix element. Thus, X in the above example can be considered as two interlaced matrices of real numbers or a single matrix of complex numbers. Operators subscripted with 1 and 2 will respectively apply to the first or second of the two "interlaced" matrices, whereas unsubscripted operators will apply to the matrix as a whole.
- Indexing of the matrix elements is from (0,0) to (m-1,n-1) where m is the number of rows and n is the number of columns.

variable – Defines a character string as a 16-byte variable outside the first 64 KB of RAM. Usage example:

```
variable Y
```

forget – Erases a matrix or variable in a manner analogous to the standard Forth word FORGET. Usage example:

```
forget X
```

dim@ – Retrieves the dimensions of the matrix and puts them on the parameter stack with the number of columns on top and the number of rows underneath. Usage example:

```
' X dim@
```

dim! – Changes the dimension of a matrix. The number of matrix elements, however, must not increase. Usage example:

```
200 1 ' X dim!
```

!1, !2, !! – Are matrix element and variable stores. !1 and !2 transfer the top number of the numeric stack to the indicated position in the matrix element or variable. !! is functionally equivalent to !2 !1. Usage examples:

```
3.14 5 15 X !1
2.72 1.23 Y !!
```

@1, @2, @@ – Are matrix element and variable fetches. Usage examples:

```
5 15 X @2
Y @1
```

EXC1, EXC – Are exchange matrix elements or variables. Usage examples:

```
5 15 X  3 2 X  EXC1
Y  8 9 X  EXC1
```

A few final notes: Because floating-point numbers and integers are stored on separate stacks, their order before an operator makes no difference. Thus, these phrases are equivalent:

```
123.456  654.321  3  7  X!!
3  7  123.456  654.321  X!!
3  123.456  7  654.321  X!!
```

Also, the use of ' should be commented on. This is called "tick" and it places the dictionary address of the word following it

onto the parameter stack. ' is for use outside a colon definition. Inside a colon definition ['] must be substituted. ' and ['] are used extensively to pass subroutines.

Mr. Ruzinsky, in an attempt to demonstrate the floating-point speed that one can achieve with Forth, brought to our attention the article "Benchmarking the 8087 Numeric Coprocessor" in the March 1983 issue of Personal Computer Age. *In it, G. Scott Owens provided some benchmarks for the IBM PC with an 8087 coprocessor (and some for the IBM and other machines without the 8087). The fastest times obtained for his floating-point routine (a mixture of various functions such as sine, cosine, and squareroot) were: IBM Pascal with an 8087 – 6 seconds; and compiled IBM BASIC with an 8087 – 10 seconds for single precision and 13 seconds for double precision. (The 8087 software in those cases was from Microware.) Mr. Ruzinsky wrote a Forth version on his own system of the floating-point routine used by Mr. Owens and obtained a time of 2.625 seconds. His Forth routine is printed in Figure 1 (above).*

```
316
 0 ( FPBENCH in Polyforth, Double Prec., 2.625 sec./500 iters.    )
 1 ( Note: SIN, COS, EX, #IN, and 4ARRAY are author written       )
 2 ( routines that are not included in standard Polyforth.        )
 3
 4 : 4ARRARY CREATE 8 * 10 + ALLOT ;CODE W INC W INC 0 POP 0 SHL
 5           0 SHL 0 SHL W 0 ADD 0 PUSH NEXT
 6              10  4ARRAY  X       LVARIABLE A
 7 : LOOP2  10 1 DO I X L@ FDUP F* PI F/ A L! LOOP ;
 8 : SBR    PI 2.1 F/ FDUP SIN 1 X L! FDUP COS
 9          2 X L!  FDUP FDUP F* 3 X L!  FDUP FSQRT 4 X L!
10          FDUP EX 5 X L! FDUP LN 6 X L!    FDUP 7 X L!
11          FDUP PI F* 8 X L!  SIN 2.0 F* 9 X L! ;
12 : FPBENCH  CR ." THIS IS A NUMBER CRUNCHING SPEED TEST "
13            CR ." ENTER THE NUMBER OF ITERATIONS " #IN
14            COUNTER SWAP 0 DO SBR LOOP2 LOOP COUNTER SWAP
15            - >N 1000.0 F/ CR ." ELAPSED TIME = " 3 F. ;
```

Figure 1.
Floating-point benchmark routine in Forth.

(Listing begins below)

Fast Matrix Operations (Text begins on page 56)

```
207 LIST

 0 VARIABLE a          CODE a! 0 POP a  STA NEXT
 1                     CODE a@ a LDA 0 PUSH NEXT
 2 VARIABLE b          CODE b! 0 POP b  STA NEXT
 3                     CODE b@ b LDA 0 PUSH NEXT
 4 VARIABLE c          CODE c! 0 POP c  STA NEXT
 5                     CODE c@ c LDA 0 PUSH NEXT
 6 VARIABLE i          CODE i! 0 POP i  STA NEXT
 7                     CODE i@ i LDA 0 PUSH NEXT
 8 VARIABLE j          CODE j! 0 POP j  STA NEXT
 9                     CODE j@ j LDA 0 PUSH NEXT
10 VARIABLE k          CODE k! 0 POP k  STA NEXT
11                     CODE k@ k LDA 0 PUSH NEXT
12 VARIABLE m          CODE m! 0 POP m  STA NEXT
13                     CODE m@ m LDA 0 PUSH NEXT
14 VARIABLE n          CODE n! 0 POP n  STA NEXT
15                     CODE n@ n LDA 0 PUSH NEXT

208 LIST

 0 ( Matrix Defining Operators )
 1 ~ VARIABLE (here) 1 (here) !
 2 ~ : here (here) @ ;
 3 ~ : forget -' ABORT" ?"  B/H - DUP 'S H 2+ @ WITHIN
 4             CAN'T H ! CURRENT CONTEXT 2+ DO I BEGIN
 5             @ DUP HERE < UNTIL I ! 2 /LOOP @ (here) ! ;
 6 ~ : allot here + (here) ! ;
 7 ~ : variable here CONSTANT 1 allot ;
 8 ~ : matrix CREATE here , SWAP 2DUP , , * allot ;CODE 0 POP
 9            1 POP 2 W) 1 ADD 4 W) MUL 1 0 ADD 0 PUSH NEXT
10 VARIABLE adr             ~ CODE adr@ adr LDA 0 PUSH NEXT
```

(Continued on next page)

```
 11                          ~ CODE adr! 0 POP adr STA NEXT
 12 ~ CODE dim@ W POP W ) 0 MOV adr STA 2 W) 0 MOV 0 PUSH
 13             4 W) 0 MOV 0 PUSH NEXT
 14 ~ CODE dim! W POP W ) 0 MOV adr STA 0 POP 0 4 W) MOV 0 POP
 15             0 2 W) MOV NEXT

209 LIST

  0 ( Matrix Element Fetch, Store and Exchange )
  1 CODE !1 0 DS SSG DS POPS R64 65520 FSTP 0 DS LSG NEXT
  2 CODE @1 0 DS SSG DS POPS R64 65520 FLD 0 DS LSG NEXT
  3 CODE !2 0 DS SSG DS POPS R64 65528 FSTP 0 DS LSG NEXT
  4 CODE @2 0 DS SSG DS POPS R64 65528 FLD 0 DS LSG NEXT
  5 CODE !! 0 DS SSG DS POPS R64 65528 FSTP R64 65520 FSTP
  6         0 DS LSG NEXT
  7 CODE @@ 0 DS SSG DS POPS R64 65520 FLD R64 65528 FLD
  8         0 DS LSG NEXT
  9 ~ CODE EXC 65520 # W MOV 0 DS SSG DS POPS R64 8 W) FLD R64
 10   W ) FLD 2 DS SSG DS POPS R64 8 W) FLD R64 W ) FLD 1 DS SSG
 11   2 DS LSG R64 W ) FSTP R64 8 W) FSTP 1 DS LSG R64 W ) FSTP
 12   R64 8 W) FSTP 0 DS LSG NEXT
 13 CODE EXC1 65520 # W MOV 0 DS SSG DS POPS R64 W ) FLD 2 DS SSG
 14   DS POPS R64 W ) FLD 1 DS SSG 2 DS LSG R64 W ) FSTP 1 DS LSG
 15   R64 W ) FSTP 0 DS LSG NEXT
```

End Listing

Yes, You Can
Trace Through BDOS

Letters to Dr. Dobb have shown an interest in using the DDT Trace function to examine the operation of BDOS functional calls. The last word on the subject (February 1983, page 11) was, "This will never work no matter how clever one is. . . ." Who could resist a challenge like that?

Demonstrating the Problem

Let's try to Trace through a very simple BDOS function and see what the problem is. BDOS function 31 is a good candidate for this because it only fetches the disk parameter address and puts it in the H/L register pair. The test program (TRACE.ASM – shown in Listing 1 below) contains the minimum elements necessary to demonstrate the problem.

(1) It initially clears the H/L pair (so we know the non-zero result was not left over from a previous run).

(2) It calls BDOS function 31.

(3) It executes an instruction (NOP) after it returns.

(4) It has an easily recognized end.

The DDT session is separated into procedures to make it easier to follow.

In Procedure 1 (page 61), I booted a 48K system (system size is important) and used DDT to get the test program in memory. The List command disassembled the object code in memory, showing that the program was actually there. The Go command executed the program (from 100 to 108) and the Examine (X) command showed that the H/L pair had B47E in it when the test program was finished. B47E is in fact the address of the disk parameter header for this particular CP/M configuration. If you run this program on your system, you will probably get a different number in the H/L pair. If you are skeptical, you can examine memory beginning at that location and verify that it is actually a disk parameter header.

The G100,100 command gets the program counter back to the beginning of the program, and T0C is the command to Trace twelve lines (DDT thinks in hex). Notice that the H/L pair was cleared, Register C was set to 31 (1F hex), and the BDOS routine was called.

by Do-While Jones

Do-While Jones, 324 Traci Lane, Ridgecrest, California 93555.

Location 0005 always contains a jump to the BDOS section of CP/M. Since the location of BDOS depends upon memory size, this address will vary depending upon system size. Whenever you boot the CP/M system, it will always make sure the jump command at 0005 is set to the proper address. In the 48K system I was using, this address was 9500. The Trace display showed the jump to 9500, and the first few instructions in BDOS, but the line following JMP A506 shows a NOP. Closer inspection shows the program counter appeared to go from 9BA7 directly to 0108. I say "appeared" because the intermediate steps really were executed. We can tell this by the fact that the H/L pair has the right answer (B47E) in it.

The invisible gap between 9BA7 and 0108 is the sequence of instructions that some readers are interested in seeing.

To Trace or Not To Trace . . .

Although it is possible to Trace through BDOS, I do not recommend that you do it routinely. DDT is designed not to Trace through BDOS functions *for your convenience.* If you defeat this feature frequently, you will just make your own life more difficult.

Inexperienced programmers use Trace at the first sign of trouble. They hit control-P, T100 and start looking through reams of paper that slowly emerge from the printer. That is not a very efficient way to debug a program, especially if it contains something equivalent to "FOR I = 1 TO 500."

The easier way to debug a program is to use break points. Use the "G" command to execute a small portion of the program. When DDT finishes executing that section, it will return with the "*" prompt, which is your cue to use "X" to examine registers or "D" to display memory. If you have written your program in modular form, there will be longer places to stop and examine the results. For example, if you have a module which is supposed to convert hex numbers into BCD form, then you can stop the program just before the routine to see what hex number is going into the routine and see if the BCD number that came out is

Simple Program for Trace Demonstration

```
                ;                       TRACE.ASM
                ;                       16 FEBRUARY 1983
                ;                       DO-WHILE JONES

                ;       THIS IS USED TO DEMONSTRATE HOW TO TRACE THROUGH
                ;       BDOS CALLS.

                ;       GLOBAL ADDRESSES

0005 =                  BDOS    EQU     0005H

0100                    ORG 100H

0100 210000             LXI H,0         ; CLEAR THE H/L PAIR
0103 0E1F               MVI C,31        ; GET ADDRESS OF DISK PARAMETERS
0105 CD0500             CALL BDOS       ; USING BDOS COMMAND 31
0108 00                 NOP             ; DUMMY OP CODE
0109 C30901     SELF:   JMP SELF        ; DO FOREVER LOOP

010C                    END
```

Listing 1.

correct. Don't Trace through the whole routine *unless* the BCD answer is wrong. Then it makes sense to Trace to see why it gave the wrong answer. If you try to use "G" to execute a portion of a routine and never see the "*" prompt, it means the program has gone to "never-never land." Then you might want to Trace to see where you PUSHed without a POP in a subroutine, or where you selected the wrong condition for a conditional jump. But it is best to try to isolate the error using break points before using Trace.

Do-While's Rule:
Never Trace more than you have to.

DDT Saves You From Yourself

The good folks at Digital Research put a lot of thought into the DDT program. They foresaw users trying to Trace through a program which contains a BDOS call to output to the console (or worse yet, read a sector). They knew how frustrated the user would be if the screen filled up with dozens of lines of BDOS code. On those rare occasions when I have resorted to Trace, and have happened to Trace through a BDOS command, I have been very grateful that DDT turned off the display until the BDOS function returned control to my own program.

Normally you should not care what the BDOS does internally. All you care about is that the return parameters are correct. You can find that out by putting a break point immediately after the BDOS call. Then you can examine registers (or memory) to see what it did, without having to try to figure out how it did it. Be sure to check the flag register. There are times when BDOS returns with the accumulator = 00, but the zero flag is *not* set. Since the JZ (Jump on Zero) instruction looks at the zero flag rather than the contents of the accumulator, your program may not branch to the desired routine unless you checked the zero flag with ANA A.

Is There Ever A Proper Time to Trace BDOS Calls?

There has only been one time that I ever found it necessary to Trace a BDOS function. I wrote an extension to CP/M 2.2 which allowed data to be displayed simultaneously on the console screen and stored in a disk file. In this case I tried to call some BDOS routines from BIOS, and the program crashed because the BDOS is not reentrant. I needed to find out why it was crashing and how to avoid it. I Traced through enough of BDOS to find out what prevented the subroutine from returning, and fortunately I was able to find other entry points which allowed me to use the BDOS subroutines to write a sector and close a file. Procedure 2 (page 62) shows the method I used.

The Trick — Two CP/M Systems at Once

After demonstrating the problem using Procedure 1, I went back to the sys-

Procedure 1.

```
48K VERSION 2.2 CP/M

A)DDT TRACE.HEX
DDT VERS 2.2
NEXT  PC
010C 0000
-L100,10B
  0100  LXI  H,0000
  0103  MVI  C,1F
  0105  CALL 0005
  0108  NOP
  0109  JMP  0109
  010C
-G100,109
*0109
-X
C0Z1M0E1I0 A=7E B=B400 D=0000 H=B47E S=0100 P=0109 JMP  0109
-
-G100,100
*0100
-T0C
C0Z1M0E1I0 A=7E B=B400 D=0000 H=B47E S=0100 P=0100 LXI  H,0000
C0Z1M0E1I0 A=7E B=B400 D=0000 H=0000 S=0100 P=0103 MVI  C,1F
C0Z1M0E1I0 A=7E B=B41F D=0000 H=0000 S=0100 P=0105 CALL 0005
C0Z1M0E1I0 A=7E B=B41F D=0000 H=0000 S=00FE P=0005 JMP  9500
C0Z1M0E1I0 A=7E B=B41F D=0000 H=0000 S=00FE P=9500 JMP  9BA2
C0Z1M0E1I0 A=7E B=B41F D=0000 H=0000 S=00FE P=9BA2 XTHL
C0Z1M0E1I0 A=7E B=B41F D=0000 H=0108 S=00FE P=9BA3 SHLD A44A
C0Z1M0E1I0 A=7E B=B41F D=0000 H=0108 S=00FE P=9BA6 XTHL
C0Z1M0E1I0 A=7E B=B41F D=0000 H=0000 S=00FE P=9BA7 JMP  A506
C0Z1M0E1I0 A=7E B=B400 D=0000 H=B47E S=0100 P=0108 NOP
C0Z1M0E1I0 A=7E B=B400 D=0000 H=B47E S=0100 P=0109 JMP  0109
C0Z1M0E1I0 A=7E B=B400 D=0000 H=B47E S=0100 P=0109 JMP  0109*0109
-G0
```

tem and PIPed the TRACE.HEX file over to another disk, which had a different size CP/M system on it. I had an old disk with a 32K system on it that I had SYSGENed when I only had 32K of memory in my system. So, *without turning off the power,* I cold-booted this 32K disk. This put a 32K CP/M system in memory, while leaving a copy of the 48K system there, too.

Procedure 2 gave results that were similar to Procedure 1, but they were not identical. The H/L pair returned the value 747E, which is exactly 16K lower than B47E because the 32K system is 16K smaller than the 48K system. Also, the JMP instruction at location 0005 sent the program to 5500 instead of 9500.

The BDOS routine at 9500 is still in memory but DDT doesn't know that. So look what happens if we change CALL 0005 to CALL 9500. We can reset the program counter to the beginning of the test program with G100,100 and Trace a bunch of lines (36 hex is enough).

This time it's all there! After the JMP A506, the program counter goes to A506, rather than skipping down to 0108.

The group of instructions up to P= A546 PCHL is the BDOS entry routine which stores parameters, sets up the BDOS stack, and jumps to the selected BDOS routine. BDOS function 31 itself is only three lines.

```
LHLD B2BB
SHLD A845
RET
```

The nine lines after that are the BDOS exit routine, which restores parameters, restores the user stack, and fetches the answer from A845.

The desired Trace is shown in Procedure 3 on page 64.

Why It Works

Using a second CP/M system does two things that permit a successful Trace of BDOS functions:

(1) DDT uses the 32K BDOS stack and 32K BDOS reserved memory locations while the test program is using the 48K BDOS stack and reserved memory locations.

(2) DDT inhibits the display when the user program counter is in the 32K BDOS, but it doesn't care that the program counter is in the reserved area of the 48K BDOS. Therefore it permits the program flow to be displayed.

There's nothing really magical about 48K and 32K. You could use any two CP/M systems, providing they don't overlap. You probably have a 20K (or 24K)

Procedure 2.

```
A)PIP B:=TRACE.HEX

32K VERSION 2.2 CP/M


A)DDT TRACE.HEX
DDT VERS 2.2
NEXT  PC
010C 0000
-L100,10B
  0100  LXI   H,0000
  0103  MVI   C,1F
  0105  CALL  0005
  0108  NOP
  0109  JMP   0109
  010C
-G100,109
*0109
-X
C0Z1M0E1I0 A=7E B=7400 D=0000 H=747E S=0100 P=0109 JMP   0109
-
-G100,100
*0100
-T0C
C0Z1M0E1I0 A=7E B=7400 D=0000 H=747E S=0100 P=0100 LXI   H,0000
C0Z1M0E1I0 A=7E B=7400 D=0000 H=0000 S=0100 P=0103 MVI   C,1F
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=0100 P=0105 CALL  0005
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=0005 JMP   5500
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=5500 JMP   5BA2
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=5BA2 XTHL
C0Z1M0E1I0 A=7E B=741F D=0000 H=0108 S=00FE P=5BA3 SHLD  644A
C0Z1M0E1I0 A=7E B=741F D=0000 H=0108 S=00FE P=5BA6 XTHL
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=5BA7 JMP   E506
C0Z1M0E1I0 A=7E B=7400 D=0000 H=747E S=0100 P=0108 NOP
C0Z1M0E1I0 A=7E B=7400 D=0000 H=747E S=0100 P=0109 JMP   0109
C0Z1M0E1I0 A=7E B=7400 D=0000 H=747E S=0100 P=0109 JMP   0109*0109
```

"distribution system" that you could use in conjunction with your full-size system.

It doesn't matter which you load first if there is sufficient space between the systems. But if the systems are nearly of equal size (like 24K and 32K), you might have trouble if you load the smaller one first. DDT moves itself to a position just below the CP/M system, so if it tries to put itself just below a 32K system, it might land on top of the 24K system. If you boot the bigger system first (as I did in the example) and then DDT from the smaller system, DDT will be below the smaller system, so it can't interfere with the larger system.

They Said It Couldn't Be Done

The previous letters to Dr. Dobb said that it is impossible to Trace in the BDOS because it is not reentrant, so "attempts to trace through BDOS will result in the saved values being overwritten and destroyed by the recursive BDOS calls made by the debugger. This tends to send you to a tight loop in never-ever land."

This is true. But just because something is impossible doesn't mean that it can't be done. Every good stage magician knows that. If you can create the illusion that it is being done, that is good enough.

Procedure 3.

```
-S106
0106 05 00
0107 00 95
0108 00 .
-L100,10B
  0100  LXI   H,0000
  0103  MVI   C,1F
  0105  CALL  9500
  0108  NOP
  0109  JMP   0109
  010C
-G100,100
*0100
-T36
C0Z1M0E1I0 A=7E B=7400 D=0000 H=747E S=0100 P=0100 LXI   H,0000
C0Z1M0E1I0 A=7E B=7400 D=0000 H=0000 S=0100 P=0103 MVI   C,1F
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=0100 P=0105 CALL  9500
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=9500 JMP   9BA2
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=9BA2 XTHL
C0Z1M0E1I0 A=7E B=741F D=0000 H=0108 S=00FE P=9BA3 SHLD  A44A
C0Z1M0E1I0 A=7E B=741F D=0000 H=0108 S=00FE P=9BA6 XTHL
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=9BA7 JMP   A506
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=A506 JMP   A511
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=A511 XCHG
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=A512 SHLD  A843
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=A515 XCHG
C0Z1M0E1I0 A=7E B=741F D=0000 H=0000 S=00FE P=A516 MOV   A,E
C0Z1M0E1I0 A=00 B=741F D=0000 H=0000 S=00FE P=A517 STA   B2D6
C0Z1M0E1I0 A=00 B=741F D=0000 H=0000 S=00FE P=A51A LXI   H,0000
C0Z1M0E1I0 A=00 B=741F D=0000 H=0000 S=00FE P=A51D SHLD  A845
C0Z1M0E1I0 A=00 B=741F D=0000 H=0000 S=00FE P=A520 DAD   SP
C0Z1M0E1I0 A=00 B=741F D=0000 H=00FE S=00FE P=A521 SHLD  A80F
C0Z1M0E1I0 A=00 B=741F D=0000 H=00FE S=00FE P=A524 LXI   SP,BE00
C0Z1M0E1I0 A=00 B=741F D=0000 H=00FE S=BE00 P=A527 XRA   A
C0Z1M0E1I0 A=00 B=741F D=0000 H=00FE S=BE00 P=A528 STA   B2E0
C0Z1M0E1I0 A=00 B=741F D=0000 H=00FE S=BE00 P=A52B STA   B2DE
C0Z1M0E1I0 A=00 B=741F D=0000 H=00FE S=BE00 P=A52E LXI   H,B274
C0Z1M0E1I0 A=00 B=741F D=0000 H=B274 S=BE00 P=A531 PUSH  H
C0Z1M0E1I0 A=00 B=741F D=0000 H=B274 S=BDFE P=A532 MOV   A,C
C0Z1M0E1I0 A=1F B=741F D=0000 H=B274 S=BDFE P=A533 CPI   29
C1Z0M1E0I0 A=1F B=741F D=0000 H=B274 S=BDFE P=A535 RNC
C1Z0M1E0I0 A=1F B=741F D=0000 H=B274 S=BDFE P=A536 MOV   C,E
C1Z0M1E0I0 A=1F B=7400 D=0000 H=B274 S=BDFE P=A537 LXI   H,A547
```

(Continued on top of page 65)

```
C1Z0M1E0I0 A=1F B=7400 D=0000 H=A547 S=BDFE P=A53A MOV  E,A
C1Z0M1E0I0 A=1F B=7400 D=001F H=A547 S=BDFE P=A53B MVI  D,00
C1Z0M1E0I0 A=1F B=7400 D=001F H=A547 S=BDFE P=A53D DAD  D
C0Z0M1E0I0 A=1F B=7400 D=001F H=A566 S=BDFE P=A53E DAD  D
C0Z0M1E0I0 A=1F B=7400 D=001F H=A585 S=BDFE P=A53F MOV  E,M
C0Z0M1E0I0 A=1F B=7400 D=0026 H=A585 S=BDFE P=A540 INX  H
C0Z0M1E0I0 A=1F B=7400 D=0026 H=A586 S=BDFE P=A541 MOV  D,M
C0Z0M1E0I0 A=1F B=7400 D=B226 H=A586 S=BDFE P=A542 LHLD A843
C0Z0M1E0I0 A=1F B=7400 D=B226 H=0000 S=BDFE P=A545 XCHG
C0Z0M1E0I0 A=1F B=7400 D=0000 H=B226 S=BDFE P=A546 PCHL
C0Z0M1E0I0 A=1F B=7400 D=0000 H=B226 S=BDFE P=B226 LHLD B2BB
C0Z0M1E0I0 A=1F B=7400 D=0000 H=B47E S=BDFE P=B229 SHLD A845
C0Z0M1E0I0 A=1F B=7400 D=0000 H=B47E S=BDFE P=B22C RET
C0Z0M1E0I0 A=1F B=7400 D=0000 H=B47E S=BE00 P=B274 LDA  B2DE
C0Z0M1E0I0 A=00 B=7400 D=0000 H=B47E S=BE00 P=B277 ORA  A
C0Z1M0E1I0 A=00 B=7400 D=0000 H=B47E S=BE00 P=B278 JZ   B291
C0Z1M0E1I0 A=00 B=7400 D=0000 H=B47E S=BE00 P=B291 LHLD A80F
C0Z1M0E1I0 A=00 B=7400 D=0000 H=00FE S=BE00 P=B294 SPHL
C0Z1M0E1I0 A=00 B=7400 D=0000 H=00FE S=00FE P=B295 LHLD A845
C0Z1M0E1I0 A=00 B=7400 D=0000 H=B47E S=00FE P=B298 MOV  A,L
C0Z1M0E1I0 A=7E B=7400 D=0000 H=B47E S=00FE P=B299 MOV  B,H
C0Z1M0E1I0 A=7E B=B400 D=0000 H=B47E S=00FE P=B29A RET
C0Z1M0E1I0 A=7E B=B400 D=0000 H=B47E S=0100 P=0108 NOP
C0Z1M0E1I0 A=7E B=B400 D=0000 H=B47E S=0100 P=0109 JMP  0109
C0Z1M0E1I0 A=7E B=B400 D=0000 H=B47E S=0100 P=0109 JMP  0109*0109
-G0
```

Julian Dates for Microcomputers

Julian dates have long been used by astronomers to facilitate accurate reckoning of dates over long periods of time, and they can also be extremely useful in various applications of computers. A Julian date is simply the number of a given day counted in sequence from some base date which is assigned Julian *date 0*. For example, if 1 January 1983 is taken as the base date 0, then 2 January 1983 would have Julian date 1, 3 January 1983 would have Julian date 2, and 1 January 1984 would have Julian date 365.

There are at least four reasons why Julian dates can be usefully applied in computer programs (inventive computerists can probably find more):

(1) The Julian date provides a compact and economical way of storing a date as opposed to representing it as a string of ASCII characters.

(2) Any two dates may be simply compared by an arithmetic test to determine which date is earlier. This is especially useful when items are to be sorted by date.

(3) The exact number of days between any two dates can be determined by a simple subtraction.

(4) When a Julian date is divided by 7, the remainder gives the corresponding day of the week (this will be explained in more detail below).

True Julian dates as used by astronomers take noon, 1 January 4713 B.C. as the base date (astronomers count days from noon to noon). But for most computer work, it's more useful to take a year someplace in the current century as a base year, because there normally is no need to reference dates very far back in the past. A base date in 1900, for example, would be sufficient for the Julian date representation of employee birth dates in almost any business *today*. Furthermore, restricting the forward span of dates from the base date to some reasonable future limit allows the Julian date to be stored and manipulated economically. For example, choosing to store a Julian date in a 16-bit word (two bytes) allows a span of 65,536 days, which is approximately 179.4 years. The algorithms for Julian date conversion given below, which use a 16-bit Julian date, allow a base year to be chosen between 1900 and 1920, thus giving a terminating date between 2079 and 2099 in the next century.

In order to use Julian dates on a computer, two conversion routines are needed: CTOJ, which converts a calendar date to a Julian date, and JTOC, which converts a Julian date back to a calendar date. The routines given here are based on Algorithm 199 in *The Collected Algorithms of the ACM*, as presented by R. G. Tantzen in 1963. Calendar dates are given in the form DAY (an integer from 1 to 31), MONTH (an integer from 1 to 12), and YEAR (an integer from 1900 to 2099). The Julian date is given (or returned) as the 16-bit unsigned integer JDATE, ranging from 0 (Julian date 0) to 65,535 (last day).

The base date is 1 March 1900, or 1 March of any leap year after 1900 not greater than 1920. The first of March rather than the first of January is used as a base date in order to avoid problems with 29 February in leap years. For similar reasons, the routines as given will not work for dates before 1900 or after 2099 (1900 and 2100 are not leap years, while 2000 is). Readers interested in representing dates outside these ranges should refer to Tantzen's original algorithms, which handle any Gregorian calendar date (but not within 16 bits).

The conversion routines are shown first in a pseudo higher-level language to make the algorithms clear (see Figure 1, below). But the reader should be cautioned that they cannot be directly translated into any higher-level language which, like many BASICs, doesn't support long (32-bit) integer arithmetic. Though the results are all 16-bit quantities (or less), some of the intermediate calculation requires 32-bit arithmetic, and the algorithms also depend on the properties of integer division.

As illustrated in Figure 1, the normal month number is converted to the number of a month in a pseudo year running from 1 March through 28 (or 29) February, and the year number is adjusted if necessary. The magic number 1461 which appears in CTOJ is simply the number of days in a leap-year cycle of three years of 365 days each and one year of 366 days. Therefore the expression (1461 * y)/4 gives the total number of days in all preceding years from the base year up to the specified year. The integer function (153 * m + 2)/5 similarly gives the total number of days in any pseudo year up to, but not including, month m. Therefore, adding DAY to the sum of the

Conversion of Calendar Date to Julian Date

```
CTOJ:   y = YEAR - Base Year
        If MONTH .gt. 2,  m = MONTH - 3
              else  m = MONTH + 9,  y = y - 1
        JDATE = (1461*y)/4 + (153*m + 2)/5 + DAY - 1
```

Conversion of Julian Date to Calendar Date

```
JTOC:   y = (4*JDATE + 3)/1461
        d = (4*JDATE + 3) mod 1461
        YEAR = y + Base Year
        d = d/4 + 1
        m = (5*d - 3)/153
        d = (5 * d - 3) mod 153
        DAY = d/5 + 1
        If m .lt. 10,  MONTH = m + 3
              else  MONTH = m - 9,  YEAR = YEAR + 1
```

Figure 1.

by Gordon King

Gordon King, King Software, P.O. Box 208, Red Bank, New Jersey 07701.

two preceding expressions gives the day number of the specified calendar date; and finally subtracting 1 gives the base-zero Julian date. The routine JTOC simply reverses the above calculation to convert back to calendar date.

The JDATE numbers used by these algorithms may also be converted to day numbers in other similar date systems by the addition or subtraction of a suitable conversion constant. Suppose that FDATE is such a day number in some "foreign" date system which also numbers days sequentially, but from a different base date. Then

$$JDATE = FDATE + CC$$

and

$$FDATE = JDATE - CC$$

where CC is the conversion constant, which is simply the JDATE value for the base date of the foreign system; it must of course lie within the valid JDATE range. For example, MP/M and CP/M Plus use a 16-bit integer date with 1 January 1978 taken as Day 1 (hence 31 December 1977 is Day 0). The JDATE value for 31 December 1977 is 28,429 if 1900 is used as the JDATE base year, or 21,124 for base year 1920. Therefore simply adding the appropriate one of these two constants to MP/M's date at the beginning of the routine JTOC converts it to the corresponding JDATE and results in the correct calendar date. Conversely, subtracting it from JDATE at the end of the routine CTOJ gives back the MP/M-form date corresponding to the given calendar date (calendar dates before 31 December 1977 will result in negative MP/M dates, which may have some useful interpretation outside of MP/M itself).

Before going on to present the two routines in Z80 assembly language, mention should be made of the method of determining the day of the week corresponding to a given Julian date. When a Julian date is divided by 7, the remainder will be one of the seven numbers from 0 to 6 (the quotient is ignored). These correspond to the seven days of the week in order, beginning from a day which depends on the base year. For base year 1900, a remainder of 0 means Thursday, 1 means Friday, and so on down to 6, which means Wednesday. For base year 1920, a remainder of 0 means Monday, 1 means Tuesday, etc. Consequently, these remainders can be used to index into a suitably ordered table of day names when it's required to print the day of the week corresponding to a given date (after it's been converted to a Julian date, of course).

Finally, the listing (page 68) provides the two conversion subroutines in Z80 assembly language. The date is transferred in the byte DAY, the byte MONTH, and the word (two bytes) YEAR, while the Julian date is transferred in the word JDATE. Since the Julian date is contained in a 16-bit word, the legal range of dates is from 1 March 1900 to 4 August 2079 if 1900 is chosen as the base year, or from 1 March 1920 to 4 August 2099 if 1920 is chosen as the base.

The subroutine CTOJ checks DAY, MONTH, and YEAR for legal values and jumps to a routine ERROR (not supplied) in case they're out of range. Two subroutines are also supplied for integer multiplication and division, but if date conversions are to be done heavily, as in an inner loop, some improvement in performance could be gained by using shifts and/or adds to do the multiplications by 4 and 5 and the divisions by 4. ■■J

```
          YEAR0    = 1900              ;BASE YEAR

;CONVERT CALENDAR DATE TO JULIAN


CTOJ:     LD       HL,(YEAR)          ;GET THE YEAR
          LD       A,(MONTH)          ;AND THE MONTH
          OR       A,A                ;CHECK FOR A LEGAL VALUE
          JP       Z,ERROR
          CP       A,12+1
          JP       NC,ERROR
          SUB      A,3                ;MONTH IN MAR-FEB "YEAR"
          JR       NC,CTOJ1
          ADD      A,12               ;JAN OR FEB BECOME 10, 11
          DEC      HL                 ;    OF PREVIOUS YEAR
CTOJ1:    LD       E,A                ;PUT MONTH IN DE
          LD       D,0
          LD       BC,-YEAR0          ;SUBTRACT THE BASE YEAR
          ADD      HL,BC
          LD       A,H                ;CHECK FOR LEGAL YEAR
          OR       A,A
          JP       NZ,ERROR
          LD       A,L
          CP       A,179+1
          JP       NC,ERROR
          PUSH     DE                 ;SAVE THE MONTH
          LD       DE,1461            ;DAYS IN A LEAP-YEAR CYCLE
          CALL     MULWW              ; DE*HL TO (DE,HL)
          LD       BC,4
          CALL     DIVLW              ; (DE,HL)/BC TO DE (QUOTIENT)
          POP      HL                 ;GET THE MONTH BACK
          PUSH     DE                 ;SAVE  (YEAR*1461)/4
          LD       DE,153
          CALL     MULWW              ; DE*HL TO (DE,HL)
          INC      HL                 ;ADD 2
          INC      HL                 ; (NO OVERFLOW POSSIBLE)
          LD       BC,5
          CALL     DIVLW              ; (DE,HL)/BC TO DE (QUOTIENT)
          POP      HL                 ;GET BACK FIRST TERM
          ADD      HL,DE              ;ADD (153*MONTH + 2)/5
          JP       C,ERROR            ;ERROR IF OVERFLOW
          LD       A,(DAY)            ;GET DAY
          DEC      A                  ;REDUCE BY ONE FOR BASE-0 J.D.
          CP       A,31               ;CHECK FOR LEGAL VALUE
```

(Continued on next page)

```
        JP      NC,ERROR
        LD      E,A             ;TO DE
        LD      D,0
        ADD     HL,DE           ;ADD IN
        JP      C,ERROR         ;ERROR IF OVERFLOW
        LD      (JDATE),HL      ;STORE AWAY
        RET

;CONVERT JULIAN DATE BACK TO CALENDAR DATE

JTOC:   LD      HL,(JDATE)      ;GET THE JULIAN DATE
        LD      DE,4            ;4*JDATE + 3
        CALL    MULWW           ; DE*HL TO (DE,HL)
        LD      BC,3
        ADD     HL,BC
        JR      NC,JTOC1
        INC     DE
JTOC1:  LD      BC,1461         ;y = (4*JDATE + 3)/1461
        CALL    DIVLW           ; (DE,HL)/BC TO DE(Q) AND HL(R)
        PUSH    DE              ;SAVE y, HL HAS d (REMAINDER)
        LD      BC,4            ;d = d/4 + 1
        LD      DE,0
        CALL    DIVLW           ; (DE,HL)/BC TO DE(Q)
        INC     DE
        LD      HL,5            ;5*d - 3
        CALL    MULWW           ; DE*HL TO (DE,HL)
        LD      BC,3
        OR      A,A
        SBC     HL,BC
        JR      NC,JTOC2
        DEC     DE
JTOC2:  LD      BC,153          ;m = (5*d - 3)/153
        CALL    DIVLW           ; (DE,HL)/BC TO DE(Q) AND HL(R)
        PUSH    DE              ;SAVE m, HL HAS d (REMAINDER)
        LD      BC,5            ;DAY = d/5 + 1
        LD      DE,0
        CALL    DIVLW           ; (DE,HL)/BC TO DE(Q)
        INC     DE
        LD      A,E
        LD      (DAY),A
        POP     HL              ;m
        POP     DE              ;y
        LD      A,L             ;MONTH = m + 3
        ADD     A,3
        CP      A,12+1          ;IF MONTH .GT. 12,
        JR      C,JTOC3
        SUB     A,12            ;     MONTH = MONTH - 12
        INC     DE              ;     y = y + 1
JTOC3:  LD      (MONTH),A
        LD      HL,YEAR0        ;YEAR = y + Base Year
        ADD     HL,DE
        LD      (YEAR),HL
        RET
```

```
;MULTIPLY HL*DE, RETURN 32-BIT PRODUCT IN DE (HIGH) AND HL (LOW)

MULWW:  LD      B,H             ;MULTIPLICAND TO BC
        LD      C,L
        LD      HL,0            ;INITIALIZE PRODUCT
        LD      A,16            ;COUNT 16-BIT MULTIPLIER
MULWW1: ADD     HL,HL           ;SHIFT DE,HL LEFT ONE
        EX      DE,HL
        ADC     HL,HL
        EX      DE,HL
        JR      NC,MULWW2       ;JUMP IF NO MULTIPLIER BIT
        ADD     HL,BC           ;ELSE ADD MULTIPLICAND TO RESULT
        JR      NC,MULWW2       ;IF LOW ORDER WORD OVERFLOWS,
        INC     DE              ;    CARRY INTO HIGH ORDER
MULWW2: DEC     A               ;CONTINUE FOR 16 BITS
        JR      NZ,MULWW1
        RET

;DIVIDE (DE,HL) BY BC. QUOTIENT TO DE, REMAINDER TO HL.

DIVLW:  EX      DE,HL           ;HIGH ORDER TO HL, LOW TO DE
        LD      A,16            ;COUNT 16-BIT QUOTIENT
DIVLW1: EX      DE,HL           ;SHIFT ONE BIT FROM DE TO HL
        ADD     HL,HL
        EX      DE,HL
        ADC     HL,HL
        SBC     HL,BC           ;WILL DIVISOR GO IN YET?
        JR      NC,DIVLW3
        ADD     HL,BC           ;NO - RESTORE HL
DIVLW2: DEC     A               ;CONTINUE FOR 16 BITS
        JR      NZ,DIVLW1
        RET
DIVLW3: INC     DE              ;INCREMENT QUOTIENT
        JR      DIVLW2          ;LEAVE RESIDUE IN HL, CONTINUE
```

End Listing

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

8088 Addressing Modes

The short comment on 8088 Base Relative Indexed Stack addressing and the table from Leo Scanlon's book, printed in this column a few months ago, drew a surprising number of letters. Albert Brunelli, of N. Chelmsford, Mass., practically wrote a whole tutorial; I found it so helpful that I am reprinting it verbatim below.

"I am writing in response to the 16-Bit Software Toolbox column of March 1983. The primary purpose of this letter is to explain some of the uses of the 8086/88's BP register. First I will comment on Leo Scanlon's listing on page 17 of the March 1983 issue.

"The subroutine will prove only that the DS register is not the default segment for BP-based memory references. The ASSUME statement does not change any register. It is merely a means of assuring the assembler that you know what you are doing when you make an anonymous reference (use a base or index as an offset without specifying a segment register). Mr. Scanlon appears to understand this when he initializes the DS register. However, he never initializes the SS register to the segment STACK which he would have to do to prove his case. If he were to change the SS to STACK, and if it weren't already pointing at STACK of course, he could never return from the subroutine. If the DL and DH registers hold 0FFh at the end of the routine, it is merely a coincidence. He would do better to push the data onto the stack and then retrieve it using the BP register. This technique will become clearer as we get further along.

"To get a good feel for the intended use of the BP register, one must understand the philosophy behind the use of the 8086 instruction set. As I understand it, Intel wanted to create a microprocessor with an instruction set which lent itself well to the implementation of high-level languages, specifically to Intel's systems language, PL/M.

"In PL/M and most other structured languages, most arguments are passed to procedures (functions, subroutines) on the stack. However, since the return address will be the lowest address on the stack, considerable manipulation must be done to access the arguments while maintaining the return address. Intel added two nice features to the instruction set to make argument retrieval easier. The BP register is one of these.

"Let's assume we wish to pass two arguments to the FAR subroutine SUB1. We might pass them as shown below:

```
        PUSH  CX
        PUSH  DX
        CALL  SUB1
```

"After the CALL instruction is executed, the stack will look like this:

```
low address   OLD IP    offset
              OLD CS    segment
              ARG2      from DX
high address  ARG1      from CX
```

"To retrieve the arguments and return properly, SUB1 might follow the procedure:

```
SUB1    PROC  FAR
        PUSH  BP
        MOV   BP,SP
        PUSH  DS      ;save registers
        PUSH  SI
        PUSH  AX
         . . . . . .  ;body of subroutine
         . . . . . .
DONE:   POP   AX
        POP   SI
        POP   DS
        POP   BP
        RET   4
SUB1    ENDP
```

"Moving SP to BP will allow us to index into the arguments passed on the stack using BP. The contents of the stack when we get to the body of SUB1 are shown as follows:

```
low address   OLD AX   ;"top" of stack
              OLD SI
              OLD DS
              OLD BP   ;BP points here
              OLD IP   ;BP+2
              OLD CS   ;BP+4
              ARG2     ;BP+6
high address  ARG1     ;BP+8
```

"We may thus retrieve the arguments with the instructions:

```
        MOV   AX,[BP+6]
                      ;fetch ARG2
        MOV   SI,[BP+8]
                      ;fetch ARG1
```

"Any reference using BP as a base will assume that it is in reference to the stack segment (with or without an ASSUME declaration).

"Now we have fetched the arguments without disturbing the return address. How will we clean up the stack so that the arguments will not clutter it up forever? Once again Intel comes to the rescue with the 'RET n' instruction. The 'n' part of the instruction tells the processor how many bytes to discard from the stack after it has taken off the return address. In other words, it will add 'n' to the stack pointer (SP) after fetching the return address.

"The final use to which I will put the BP register is one which I have found very helpful in memory-test programs which do not have access to the, as yet unverified, stack area of RAM. We may create synthetic subroutines which may be

```
                LEA     BX,TABLE_1          ;point to ROM table
                MOV     CX,NUM_ENTRIES      ;number of table entries
                LEA     BP,CONT1            ;BP becomes return addr
                JMP     SYN_SUB1
CONT1:          . . . . . .                 ;control returns here
                . . . . . .

SYN_SUB1;                                   ;synthetic subroutine
                . . . . . .
                . . . . . .
                JMP   BP                    ;pseudo-return instr.
```

Figure 1.

'called' from anywhere within the same code segment. The idea is that the return address is placed in a register before jumping to the subroutine, which terminates with a jump register instruction such as shown in Figure 1, page 72.

"The LEA BP instruction will place the offset of the CONT1 label in the BP register. The BP register is the perfect one to use in a situation like this because we have no stack to which it may point. The JMP BP instruction may be thought of as roughly equivalent to the 8080's PCHL instruction, although it is more powerful since any general register may be used as the source."

Wishful Thinking Dept.

Jim Howell wrote to point out a truly glaring error, found on page 3-13 of the *iAPX-88 Book* (July 1981 edition), which readers of this column may find amusing or amazing. "16-bit operands are stored in memory with the most significant byte (MSB) first, followed by the least significant byte (LSB) in the next location." The picture set at the bottom of the page in the book repeats this error.

8088 Line Generator

Dan Rollins of Glendale, Calif. sent in an 8088 assembler version of the fast-vector algorithm featured in Dave Cortesi's column of December 1982 and February 1983. His comments follow:

"The routine is self-modifying and needs no external storage area. The code that calculates the coordinate pairs is blazing fast. It's a shame that the BIOS write-dot routine (which it calls) is so slow. I am using a version of this routine in an arcade game I am currently writing. All of the drawing and point testing in that game takes place in a buffer and I use a much faster PLOTDOT routine."

Listing 1 (page 75) contains the line generation subroutine proper, while Listing 2 (page 77) demonstrates how to call the subroutine from BASIC.

PC-TALK III

Andrew Fluegelman, author of the excellent PC-TALK communications program and originator of the "Freeware" concept, has announced a new version that has some significant enhancements. Version III has been rewritten so that all features operate at 1200 baud. The Key Directory has been expanded to 40 permanent strings, and the Dialing Directory now holds 60 entries and stores selective character stripping for each entry.

The Transmit and Receive routines now offer the options of line-paced transmission, sending of binary files, and the XMODEM error checking protocol. Program defaults can be reset interactively while the program is running. Other new bells and whistles include auto-redial, margin width alarm, and a screen dump to disk.

Updates to Version III for previous PC-TALK owners are only $10.00, while the suggested donation for new owners is $35.00. As before, everyone is encouraged to make copies of the program and distribute them as widely as possible for free; recipients of such copies are requested to mail in a donation if they appreciate the program and want to promote the development of more Freeware. PC-TALK is available from The Headlands Press, P.O. Box 862, Tiburon, California 94920.

(Listings begin on page 75)

The Line Generator Subroutine

Listing One (Text begins on page 72)

```
                                page ,132
                                title DRAWLINE.ASM      Dan Rollins

                                public drawline

                        ;8088 self-modifying program implements fast-vector algorithm
                        ; described by Michalsky, DDJ #74, 12/82
                        ; see also: FAST-LINE DRAWING TECHNIQUE, BYTE, Aug 81

                        ;routine expects to be called from BASIC via:

                        ;       CALL DRAWLINE(VX(0))

                        ;  where VX(0)   = X1    starting clm (0-319)
                        ;        VX(1)   = Y1    starting row (0-159)
                        ;        VX(2)   = X2    ending clm
                        ;        VX(3)   = Y2    ending row
                        ;        VX(4)   = color  (0,1,2,3)
                        ;        VX(5)   = length
                        ;                  0    = draw entire line
                        ;                  else = draw sub- or super-set of this vector

                        code    group   cseg
0000                    cseg    segment public 'code'
                                assume  CS:cseg, DS:nothing, ES:nothing

                        ; make it easier to access variables and arguments
=                       ARG1    equ     word ptr [BP+6]

=                       X1      equ     word ptr [si]
=                       Y1      equ     word ptr [si+2]
=                       X2      equ     word ptr [si+4]
=                       Y2      equ     word ptr [si+6]
=                       COLOR   equ     byte ptr [si+8]
=                       LEN     equ     word ptr [si+10]

                        ; these are values that will be inserted in the code
= 0041                          INC_X   equ     41H
= 0049                          DEC_X   equ     49H
= 0042                          INC_Y   equ     42H
= 004A                          DEC_Y   equ     4AH

                        ; these are the addresses where new code is overlayed
=                       ADJ_LONG_AXIS   equ   byte ptr cs:[di]
=                       ADJ_MASTER      equ   word ptr cs:[di+3]
=                       TEST_MASTER     equ   word ptr cs:[di+7]
=                       ALT_ADJ_MASTER  equ   word ptr cs:[di+13]
=                       ADJ_SHRT_AXIS   equ   byte ptr cs:[di+15]

                                page
0000                    drawline proc   far
0000  55                        push    bp        ;always save
0001  8B EC                     mov     bp,sp

0003  8B 76 06                          mov     si,ARG1   ;si => address of X1 ;ie, VX(0)
```

```
0006  B3 41                          mov     bl,INC_X  ;assume Xstep = +1
0008  8B 44 04                             mov     ax,X2
000B  2B 04                          sub     ax,X1
000D  7D 04                          jge     d11       ;if X1 <= X2 then no change

000F  B3 49                          mov     bl,DEC_X  ;Xstep = -1
0011  F7 D8                          neg     ax        ;Xdist = abs(Xdist)
0013                            d11:
0013  8B C8                          mov     cx,ax     ;save Xdist

0015  B7 42                          mov     bh,INC_Y  ;assume Ystep = +1
0017  8B 44 06                             mov     ax,Y2
001A  2B 44 02                             sub     ax,Y1
001D  7D 04                          jge     d12       ;if Y1 <= Y2 then no change

001F  B7 4A                          mov     bh,DEC_Y  ;Ystep = -1
0021  F7 D8                          neg     ax        ;Ydist = abs(Ydist)
0023                            d12:
0023  8B D0                          mov     dx,ax                     ;save Ydist
0025  BF 005D R                      mov     di,offset cs:modify_base ;point to the code
                                                                       ; to modify
0028  3B D1                          cmp     dx,cx                     ;determine longest axis
002A  7D 04                          jge     d13                       ;Y is longer, so skip

002C  87 CA                          xchg    cx,dx             ;swap Xdist, Ydist
002E  86 DF                          xchg    bl,bh             ;swap INC/DEC X/Y values
0030                            d13:                           ;modify:
0030  2E: 88 3D                      mov     ADJ_LONG_AXIS,bh  ; the 1st INC/DEC code
0033  2E: 89 4D 03                   mov     ADJ_MASTER,cx     ; main duty master adjustment
0037  D1 E9                          shr     cx,1              ;set up cycle tester
0039  2E: 89 4D 07                   mov     TEST_MASTER,cx    ; test for cycling
003D  2E: 89 55 0D                   mov     ALT_ADJ_MASTER,dx ; alternate adjustment
0041  2E: 88 5D 0F                   mov     ADJ_SHRT_AXIS,bl  ; alternate INC/DEC code

0045  8B FA                          mov     di,dx          ;DI is counter: long axis length
0047  83 7C 0A 00                   cmp     LEN,0          ;if length arg > 0
004B  7E 03                          jle     d14
004D  8B 7C 0A                             mov     di,LEN         ; then use it as counter
0050                            d14:
0050  8B 0C                          mov     cx,X1
0052  8B 54 02                             mov     dx,Y1
0055  8A 44 08                             mov     al,COLOR
0058  33 DB                          xor     bx,bx          ;duty master starts = 0
                                      page  ;---------- top of vector plotting loop ---------
005A                            d15:
005A  E8 0074 R                      call    plotdot        ;plot a dot

005D                            modify_base     label   byte

005D  41                             inc     cx        ;INC/DEC CX/DX:  adjust long axis ptr
005E  81 C3 1111                     add     bx,1111H  ;Xdist or Ydist: adjust duty master
0062  81 FB 1111                     cmp     bx,1111H  ;Ydist or Xdist: check cycle position
0066  7E 05                          jle     d16       ;skip if short axis is still ok
```

(Continued on next page)

Calling the Subroutine from BASIC

```
0068  81 EB 1111                    sub     bx,1111H  ;Xdist or Ydist: adjust duty master
006C  42                            inc     dx        ;INC/DEC DX/CX:  adjust short axis ptr
006D                          d16:
006D  4F                            dec     di        ;di is used as counter
006E  7D EA                         jge     d15       ;do next dot if not finished
                                    ;-----------------------------------------

0070  5D                            pop     bp        ;always restore
0071  CA 0002                      ret     2         ;back to BASIC, discard 1 arg <<EXIT<<
0074                          drawline endp

                              ;this routine plots the pixel at column CX  (0-319 or (0-639)
                              ;                                row    DX  (0-199)
                              ;                                color  AL  (0-3)  or (0-1)
0074                          plotdot proc    near
0074  50                            push    ax
0075  57                            push    di              ;BIOS destroys these registers

0076  B4 0C                         mov     ah,12           ;write_dot function
0078  CD 10                         int     10H             ;  video I/O call

007A  5F                            pop     di
007B  58                            pop     ax
007C  C3                            ret
007D                          plotdot endp

007D                          cseg    ends
```

End Listing One

Listing Two

```
' TESTER.BAS          Dan Rollins  02/02/83

' this compiled BASIC program tests the 8088-code DRAWLINE routine

  defint a-z
  dim v%(5)
  screen 1
10:
  input"routine (1 or 2)";r
  if r=2 then 100

20:
  input"color (0-3, -1 to quit)";v%(4)
  if v%(4) < 0 then 30

  input"x1,y1,x2,y2";v%(0),v%(1),v%(2),v%(3)

  v%(5)=0
  input"length (null = full line)";v%(5)

  call drawline(v%(0))

goto 20

30:
   screen 0,0,0,0 :width 80 :end

'-----------------------------------------
```

Listing Two (Listing continued, text begins on page 72)

```
' LASER BEAMS:  sample use of the length parameter

100:
    cls
    vX(0)=159 :vX(1)=(199)     'X1,Y1 = bottom center of screen
110:
    vX(2)=int(rnd*312)         'X2    = random column
    vX(3)=0                    'Y2    = top of screen

    vX(4)=3                    'color = white
    for length=10 to 190 step 10
       vX(5)=length            'set length
       call drawline(vX(0))    'draw partial line
    next
    vX(4)=0                    'color = black to erase
    for length=10 to 190 step 10
       vX(5)=length
       call drawline(vX(0))
    next

if inkey$="" then 110 else 30 'any key to exit
```

End Listing Two

CP/M EXCHANGE

by Robert Blum

Take a Look in the Public Domain...

Public domain software libraries continue to grow and the number of computerists involved in related activities is virtually exploding. The two predominate groups responsible for cataloging and distributing public domain software are the CP/M Users Group (CPMUG) and the Special Interest Group for Microcomputers (SIG/M). Collectively they offer over 140 volumes (almost 23 Mbytes) crammed full of programs. Many smaller groups specializing in specific programming languages and machines exist. Most have program libraries and are responsible for many of the software contributions of the larger groups.

CPMUG (1651 Third Avenue, New York, New York 10028) is the single largest distributor of CP/M public domain software, with 85 volumes in their library. No matter what your needs, from utilities to BASIC business software, it can be found in their catalog. At a nominal cost of $12 per volume, a better bargain can't be found. Unfortunately, one stumbling block exists. With each new computer comes a new disk format and stocking all of them is beyond the resources of most groups. In response to this need, a few commerical concerns are now offering copies of the original disks in many of the more popular disk formats.

What is desperately needed is some universal form of machine-readable media. Since none are currently available, two alternate approaches can be used. The least costly is "paperware." Even though the cost is low, many hours will be spent keying in the program and then finding the transcription errors. The second approach requires the purchase of a modem. With a 300 baud modem you will have ready access to a number of Remote CP/M (RCPM) systems across the country. Many of them offer complete libraries of public domain software in addition to an amazing number of other useful programs.

One often-overlooked point is the educational value of public domain software. Many fine books and reference documents are available to provide the prerequisite training on the computers instruction set and the CP/M interface. But none can rival the first-hand experience of stepping through a working program with a debugger. As if by magic, the mysteries are solved as you watch each instruction execute.

If a list of the foremost activists in public domain software were made, Ward Christensen would rank in the top ten. His involvement with microcomputers began with their inception. As librarian of CPMUG, he exemplifies dedication to the public domain scene. To ease the log jam of phone calls to his RCPM system he has added a second phone line as an "administrative" and general CP/M message system (messages deal with bug fixes, reviewing and cataloging of CPMUG disks, etc.). Use 312-849-1132 (and the new "M" command and "CPMUG" password) to download the current CPMUG catalog of programs (not the individual files), retrieve a copy of the user group contribution form, send in a contribution, etc. The original CBBS in Chicago is at 312-545-8086, which supports baud rates of 110-600. Press return several times for speed detect. When using these numbers, be patient – they get a lot of activity.

Periodically, I will be reviewing programs from the public domain that provide topical information. For the edification of those who are interested, I will offer complete program listings along with the review at my cost of printing and shipping.

...at SD-44

Displaying the diskette directory is probably one of the most-used functions of CP/M. Unfortunately, the built-in command DIR produces an unsorted, single-column display which is hardly satisfactory when several hundred file names are to be scanned. CPMUG volume 85 contains SD-44, which I believe is the latest version in a series of full-feathered directory programs. The display produced is sorted, and three columns wide. Each file's size is given and the total space remaining on the diskette is calculated. This may sound like a description of many others but SD-44 offers a number of other features. The eight options recognized are:

S – system option: includes files in the output.

F – file option: echoes the directory output to a disk file on the default drive, named "SD.DIR." If SD.DIR already exists, then the directory output will be *appended* to the end of the file. Otherwise, SD.DIR will be created as a new file.

U – user option: allows the specification of the user number for the directory in the form "U*n*" where the user number, *n*, must be greater than 0 and no larger than 15. This option allows no spaces between "U" and *n*, and cannot be used on pre-CP/M 2.0 systems.

A – all users: displays directories of all user areas starting at the user area specified in the U option. If the U option is omitted, displays start with the default user area and continue up to MAXUSR.

R – reset option: provides automatic resetting of the disk prior to performing directory seach, updating the allocation vector. Same as doing a Ctrl-C when changing disks, but handy if you didn't (such as when running a SUBMIT file). The RESFLG equate will force the R option unconditionally each time SD is run.

N – no page option: unconditionally disables the page pause option. SD will not put the page-pause prompt into the output file.

P – printer option: forces all console output to be echoed to the CP/M list device, with the most significant bit set to 0.

D – all disk option: allows SD to search all disk drives online starting with the disk drive specified or implied in the command line filename.

Outside of the obvious usefulness of SD-44, most of the techniques described in my series on BIOS internals can be found in this program. If you want a paperware copy of SD-44 and its documentation, send $5.00 to the address at the end of this column.

More on CP/M Plus

The real *story behind* CP/M Plus's features unfolds when the BDOS functions are examined. Rather than include all the BDOS functions, Table I (page 82) only lists the additions and changes.

Many of the BDOS functions are now MP/M compatible. This fact makes me wonder what DR has in mind for future releases; possibly adding concurrent operation to CP/M or providing a natural upgrade path to MP/M. Whatever the

case, to make full use of all the new features will require some changes to the application program. Whether CP/M 2.2 programs will run without change under CP/M Plus is a question that can best be answered by running it. Providing trick code has been kept to a minimum, the success ratio should be high.

Next month I will continue with memory mapping and more on CP/M Plus's internals. You can reach me directly if you want to order paperware for SD-44, or to discuss subjects from this column, at: Bob Blum, 5536 Colbert Trail, Norcross, Georgia 30092; (404) 449-8948.

DDJ

Table I.

CP/M Plus BDOS function addition and changes.

3 – *Auxiliary Input:* same as CP/M 2.2 reader input.

4 – *Auxiliary Output:* same as CP/M 2.2 punch output.

7 – *Auxiliary Input* status: new function call to test whether data is available at the auxiliary input device. Under CP/M 2.2 was *Get I/O Byte.*

8 – *Auxiliary Output* status: new function call to test whether data can be sent to the auxiliary device. Under CP/M 2.2 was *Set I/O Byte.*

38 – *Access Drive:* MP/M function call, provided for compatibility only.

39 – *Free Drive:* MP/M function call, provided for compatibility only.

41 – *Test and Write Record:* MP/M function call, provided for compatibility only.

42 – *Lock Record:* MP/M function call, provided for compatibility only.

43 – *Unlock Record:* MP/M function call, provided for compatibility only.

44 – *Set Multi-Sector Count:* allows from 1 to 128 128-byte sectors to be read or written in one operation. MP/M compatible.

45 – *Set BDOS Error Mode:* determines how errors are handled. MP/M compatible.

46 – *Get Disk Free Space:* determines how many free 128-byte sectors there are on the specified drive. MP/M compatible.

47 – *Chain to Program:* allows automatic chaining from one program to another. MP/M compatible.

48 – *Flush Buffers:* forces any remaining records marked for writing to be written. MP/M compatible.

49 – *Get/Set System Control Block:* allows the system control block to be changed.

50 – *Direct BIOS Calls:* allows direct BIOS calls to be made through BDOS functions. CP/M Plus no longer supports direct BIOS calls.

59 – *Load Overlay:* load a Resident System Extension (RSX) or overlay into memory.

60 – *Call Resident System Extension:* special function for loading RSXs only.

98 – *Free Blocks:* returns to free space any blocks that have been allocated, but not permanently recorded on disk.

99 – *Truncate File:* truncates the specified file to the indicated random record number.

100 – *Set Directory Label:* creates or updates the directory label which indicates what extended directory options are active. For example, perform access date and time stamping. MP/M compatible.

101 – *Return Directory Label Data:* returns the directory label data byte. MP/M compatible.

102 – *Read File Date Stamps and Password Mode:* returns password and time stamp mode for specified file. MP/M compatible.

103 – *Write File XFCB:* creates or updates the XFCB for a specified file. MP/M compatible.

104 – *Set Date and Time:* sets the internal date and time. MP/M compatible.

105 – *Get Date and Time:* returns the internal date and time. MP/M compatible.

106 – *Set Default Password:* allows setting of a file password before a file is accessed. MP/M compatible.

107 – *Return Serial Number:* returns the 6-byte CP/M Plus serial number.

108 – *Get/Set Program Return Code:* allows a program to set a termination code for access by other programs which may follow.

109 – *Get/Set Console Mode:* allows setting of control parameters for certain BDOS console functions.

110 – *Get/Set Output Delimiter:* allows the string delimiter used in function 9 to be set to another value.

111 – *Print Block:* print the block of data pointed to by a CCB on the console.

112 – *List Block:* print the block of data pointed to by a CCB on the system list device.

152 – *Parse Filename:* parse a filename and prepare a file control block for use.

OF INTEREST

by Michael Wiesenberg

Impressions of the West Coast Computer Faire

It seems *de rigueur* for computer columnists to chronicle their impressions of the Computer Faire, so here are mine. As you all know, the West Coast Computer Faire, run by Jim Warren (one of *DDJ's* first editors), happens each spring at San Francisco's Civic Auditorium and Brooks Hall, with symposia and workshops at the San Franciscan and Holiday Inn hotels. Many call the Faire the most important microcomputer conference of the year.

It was more crowded than last year. Also more hype. The most impressive display was that of **Perfect Software**, obviously designed with great thought by marketing folks. Some computer businesses are just beginning to realize that good products don't necessarily sell themselves; they have to be *sold*. The Perfect folks used the principle well, as they lured you up a carpeted ramp lit on either side by flashing bulbs synchronized to simulate a lighted path travelling into the darkened interior recesses wherein brilliant blue laser blasts pierced clouds of smoke. Displays and booths everywhere else in the Faire were out in the open and jammed up against the wares of other companies; but PS had, probably at considerable expense, bought a large corner nook and turned it into a mysterious subterranean cave. As you stepped off the ramp, your eyes slowly adjusted to the gloom and you found yourself on a railed platform from which two staircases descended into a maze of monitors on pedestals, all displaying Perfect Software products. Giant frameworks of steel girders supported laser-emitting devices that randomly shot blue pencils of light above the heads of the crowd, creating moire patterns in clouds of dry-ice steam. A disembodied, amplified, sepulchral voice, describing the virtues of various Perfect products, floated through and around the din of the crowd.

Elsewhere, **Texas Instruments** put on a robot show. A wheeled contraption carrying a TI 99/4A on a tray, having a TV lens for a face and a monitor mirroring all it saw where its mouth belonged, wandered through the crowd near the TI booth, trading wisecracks with a "real" person dressed up as Charlie Chaplin (and symbolizing, perhaps, the dominance of the TI machine over IBM?) and carrying on apparently intelligent conversations with bystanders. I am familiar enough with robotics to know that this "robot" must have been controlled at a distance by human beings. I looked around for, but could not find, some seemingly innocent person perpetually draining a coffee cup, but in actuality speaking into a microphone concealed therein.

Both software and hardware were on sale for greatly reduced prices. If you know precisely what you're looking for, computer fairs probably have the best bargains anywhere. But, *caveat emptor:* not all of the exhibitors will be back next year.

The machine that impressed me the most was the Dynalogic **Hyperion**, the truly portable, beautifully designed 16-bit system that I described last month. The version I saw, the Hyperion Plus, costs close to $5000, but you can get one without quite so many bells and whistles for under $3400.

The machine that may well start a new trend in quality low-cost computing is the **Humdinger**. This tiny Z80 color computer with CP/M (described in detail below) costs but $129 yet offers features found only in computers costing literally thousands more.

Ah, yes, then there's **Lisa**. Wonderful machine. Maybe the best user interface for a micro commercially available. The mouse is easy to use, moves quickly anywhere on the screen, and instantaneously displays information any way you want it, without having to touch the keyboard. But $10,000? If they want to compete in the same ballgame with the PC, Apple will have to price Lisa in the same ballpark.

Through all the chaos glided Jim Warren on roller skates, constantly coordinating his show through a walkie-talkie that rarely left his lips.

In addition to manufacturers with products to display, acquisition editors for major publishers kept an eye out for potential authors of promising material, and software makers were on the lookout for new programs. If you have a book or software to sell, you'll find buyers at the Faire.

On Sunday at 5 p.m., the sated crowds were quickly hustled out into some of the worst rain San Francisco had experienced in years.

I was suffering from sensory overload, but glad I had attended. The same thing happens when I spend more than a few hours at a first-rate art museum.

Hard PHD or Toaster for Your System

The **Computer Service Company** offers 16 Mb Winchesters for most systems for $2595, 8 Mb for $2195, and double 5Mb removable subsystems with two free cartridges for $2795. These are variously called **PHD 8x8, PHD 4x4, PHD 8x8KP** (for KayPro, for example), **PHD 8x8S** (S-100), all of which are 5¼-inch drives, and **Toaster** (two 3.9-inch drives). The systems include drive, parallel interface, Z80 adapter (or adapters for virtually any other computer), transportability between computers, power supply, diagnostics, format, sector sparing program, driver, six-month warranty, and (here's the best part) *free* installation by The Computer Service Company, FOB Mountain View, California. While I'm giving them a mention, The Computer Service Company also rents all kinds of computer equipment, from Osbornes at $5 a day, dot matrix printers at $2.50, floppy disk drives at $2.50, hard disks at $7.50, to monitors at $1.25 (add about one-third to include service), and they offer phone consultation on all forms of hardware and software maintenance and design for $35 per hour. **Reader Service No. 101.**

Graphics Like A Word Processor

Graphics Processing System, for 48K Apple II Plus, from **Stoneware**, manipulates and edits images like a word processor. It will also mix and change colors at will, edit or erase a portion of a picture merely by defining its boundaries, zoom or reduce any *portion of a picture four or 16 times* (with the stored image having a greater resolution than that on screen, reproducible depending on the capabilities of the printer or plotter), rotate images in two dimensions, duplicate images on screen and to and from disk, change proportionality of portions or all of images, overlay in separate colors, and access the 16K RAM card. The system is compatible with graphics tablets,

light pens, and various plotters and graphics printers. $179. **Reader Service No. 103.**

A Good Case

Comp Cases by the **Computer Case Company** make almost any small system portable, inexpensively. Literally a custom built suitcase that exactly fits your IBM PC, Apple, TRS-80, etc., various monitors, printers, and other accessories, the Comp Case is built from mahogany plywood, has padded handles, brass hardware, key locks, vinyl cover, triple-thick, saddle-stitched vinyl at the edges, rubber feet, custom interior foam padding, nylon velcro straps, and interlocking top and base. Once your system is in a Comp Case, you need never remove it, so you can leave cabling in place and easily set up the system. Comp Cases vary in price from less than $100 for many peripheral or accessory cases, to $109 for a housing for Apple II with one drive, to $129 for Apple II with two drives and a monitor or TRS-80 Model III, to $139 for Apple III with drives and printer. **Reader Service No. 105.**

A Real Humdinger

The **Humdinger** from **Venture Micro** is a Z80 color computer with a "real" keyboard, 4K RAM, 8K BASIC in ROM, eight-color video, four-voice sound generator, free game cartridge, RF modulator, and parallel, serial, cassette, cartridge, joystick, expansion, and EPROM interfaces, for $129. You can add 16K RAM for $39.95, 64K for $99, voice synthesizer for $69.95, disk controller for $75, 5¼-inch drive for $210, CP/M for $79, word processor for $45, Pascal for $59, an 8088 for $119 (you'll also need the 8088 BASIC ROM for $124.95), and an 8087 for $299, travel case, graphics table, 80-by-24 video, user-defined graphics, real time clock calendar, editor/assembler, various game cartridges and cassettes, extended BASIC, COBOL, Forth, C, Logo, and Pilot. Ten new cartridge and cassette programs per month are planned. **Reader Service No. 107.**

Dot Matrix Printer with Daisy Wheel Quality

The **Santec S700 Printer** from **Western Technology** offers letter quality by printing each line four times with minute advances between passes, at a throughput of 32 to 58 cps. Since it prints 960 dots per inch, you'll need a magnifying glass to distinguish this mode from a daisy wheel output. A correspondence mode makes two passes at each line, with quality better than that of the best dot matrix printers, at a throughput of 65 to 195 cps. And the draft quality mode is as good as the best dot matrix printers. Up to 12 fonts, ranging in size from 7 to 12 points, can be intermingled simultaneously, with no pause in printing. Formatting commands, inserted in the text, are performed by the printer, rather than the computer, with total user control over margins, spacing, tab settings, justification (incremental and proportional), centering, boldface, underlining, and graphics. This remarkable printer costs under $4000. Spellbinder, the CP/M word processing software from Lexisoft, has a special version configured for the Santec. **Reader Service No. 111.**

Two APLs

VIZ::APL from **EASI APL Systems** is virtual-memory APL for Z80 microcomputers, and some 16-bit microcomputers with a Z80 board. This full implementation of APL has an overlaid interpreter and virtual work area limited only by disk space, runs under CP/M, interfaces with high-resolution graphics, uses double-precision, floating-point arithmetic, and has dynamic symbol table allocation. I saw the language demonstrated on an Osborne I at the West Coast Computer Faire, and was impressed with its fast, direct-mode vector calculations and transformations. No price on this one, but use the Bingo Card for more information. **Reader Service No. 115.**

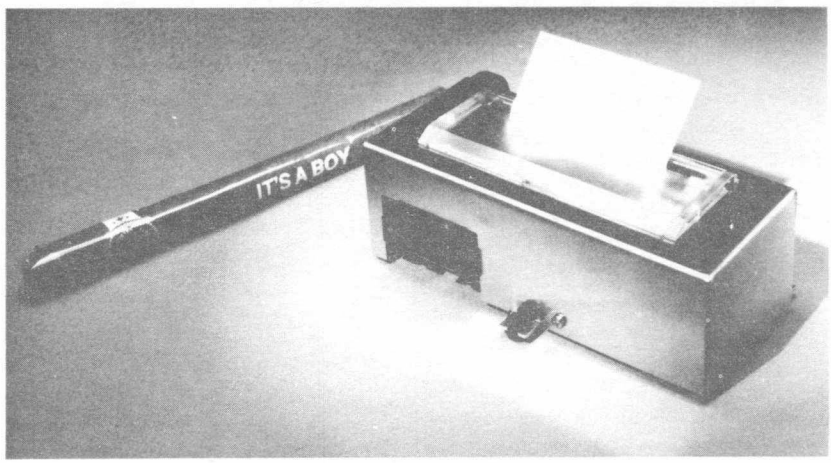
APL.68000 by the **Computer Company** is a full APL that runs under CP/M-68K, UNIX, and VersaDOS on the Sage Models 2 and 4, Forward Technology's Gateway Work Station microcomputers, the Pixel 100AP, the Corvus Concept, and "soon" on all 68000-based micros. **Reader Service No. 117.**

1/4

The **Forth-79 Version 2** compiler from **MicroMotion** for most Z80 CP/Ms 1.4 and 2.whatever, including Apple, comes with screen editor, macroassembler, string processing, three-bit arithmetic, floating point, 200 pages of tutorial and reference documentation, and hi-res for Apple and North Star, and costs from $99.95 (doesn't anything ever sell for an even $100?) to $139.95. **Reader Service No. 119.**

Wooden Clay?

Perhaps one of the better pieces of inexpensive software for the IBM PC seen at the West Coast Computer Faire was **Cosmic Nightware** from **Wood & Clay Hi-Tech Gameware.** "Earth explodes," they say (I saw it



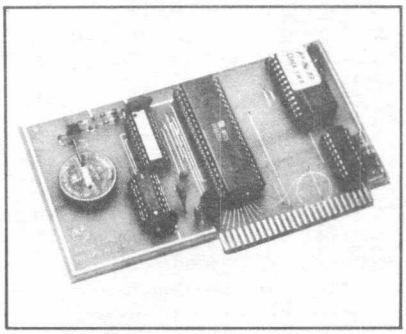
Hot Little Printer

The **EUY-3T** thermal printer from **Panasonic** is about the size of a good 5-cent cigar ("What this country needs is a good . . ." inexpensive printer), 4.69 by 1.79 by 2.64 inches. With dot-addressable graphics, 40 characters per line, battery power, bi-directional printing at 1.2 lines per second, and a weight of 14 ounces, the printer will cost under $100 in OEM quantities. **Reader Service No. 109.**

happen – wonderful graphics), "and you're the only one left! All the phantoms of the universe are released, and you have to fight enemies so real you won't believe it's a dream." The disk costs $32.50. If you want real arcade action on your PC, you'll want to order joystick controllers at $29.95 each from W&C. Other games available include **Falklands Fury**, with "all the action of real life warfare" (whoopee!, what fun!) and **Jungle Madness**, in which you "play Tarzan, swing from vines, and fight off savage pygmies," also $32.50 each. (California, add 6.5% sales tax.) **Reader Service No. 121.**

The Sorcerer Lives!

I often think my computer system is unusual and incompatible with anyone else's. When I say I have a Sorcerer, most people say, "A what?" So I'm always pleased to find someone trying to keep this wonderful Z80 machine alive. In this case, it's **ISIS** (the International Sorcerer Information Service), a "not-for-profit" monthly newsletter to give Sorcerer owners a means of exchanging information. Membership is $15 in Canadian funds in Canada, and US dollars elsewhere. Contact ISIS, c/o Maurice Dow, 84 Camberley Cres., Brampton, Ontario, Canada L6V 3L4; or use the reader service card. **Reader Service No. 123.**



Clock Your Apple

dat.a.clock, for Apple II, II Plus, and IIe, from **P & B Research Consultants** keeps time in date, month, and year, with a two- to three-year battery, has an externally accessible EPROM, and costs $55 in kit, or $85 assembled (plus $2 p&h). **Reader Service No. 113.**

Commodore Camaraderie

A VIC-20 users group is being formed by the **New York Amateur Computer Club**. Although the UG will meet in New York City, NYACC is soliciting members throughout the country to join "on a correspondence basis." Both types of VIC-20 users are sought, computer novices and the knowledgeable who bought the machine "just for the fun of it. What we would like to do is put these two types together and get some synergism going." If enough interest is generated, NYACC will plan to expand to other Commodore machines, like the 64 and the Pet. **Reader Service No. 125.**

Make a Beeline for the C Line

The C Line, a free public access electronic bulletin board for users of UNIX and C, has close to two megabytes of public domain C software just itching to be downloaded onto the computers of those with any standard terminal or computer with modem who dial (201) 625-1797, 8 pm to 9 am weekdays (Eastern time) and 24 hours weekends. The C Line, say sponsors **In-foPro Systems**, publishers of *UNIQUE*, the UNIX industry newsletter, and Perchwell Corporation, a management research/consulting firm specializing in UNIX, "expects to receive several carriage returns initially" (ah, anthropomorphisms abound!), "and will automatically adjust itself to the transmission speed being used, from 110 to 710 baud." **Reader Service No. 129.**

Confer Intelligently

The National Conference on Artificial Intelligence, sponsored by the American Association for Artificial Intelligence, takes place August 22 to 26, 1983, at the Washington (D.C.) Hilton Hotel. **Reader Service No. 131.**

Keeping TABs

TAB Books' Spring/Summer catalog, with 65 new books and over 600 titles, includes these about computers:

Graphics Programs for the IBM PC (256 pages, $12.95 paper, $18.95 hardcover), *33 Games of Skill and Chance for the IBM PC* (256/$14.95/$21.95), and, possibly a good one for would-be hardware tech writers, *Beginner's Guide to Reading Schematics* (140/$8.95/$13.95), all by Robert Traister.

100 Ready-to-Run Programs and Subroutines for the IBM PC, by Jeff Bretz and John Craig (320/$15.95/$22.95).

Programming Your Atari Computer, by Mark Thompson (280/$10.95/$16.95).

Advanced Programming Techniques for Your Atari, including Graphics and Voice Programs, by Linda Schreiber (224/$13.95/$19.95).

25 Graphics Programs in Microsoft BASIC, by Timothy O'Malley (160/$10.95/$17.95).

101 Projects for the Z-80, by Frank Tedeschi and Robert Colon (368/$16.95/$23.95).

Troubleshooting and Repairing Personal Computers, by Art Margolis (320/$13.95/$19.95).

Microcomputer-Controlled Toys and Games and How They Work, by Van Waterford (240/$9.95/$17.95).

Experiments in Four Dimensions, by David Heiserman (288/$12.95/$21.95).

Learning Simulation Techniques on a Microcomputer Playing Blackjack and Other Monte Carlo Games, by Pat Macaluso (154/$10.95/$16.95).

Basic BASIC-English Dictionary for the Apple, PET, and TRS-80, by Larry Noonan (154/$17.95, hard cover only).

The Sinclair ZX81, by Randle Hurley (182/$16.95, hard cover only).

Electronic Components Handbook for Circuit Designers, by R.H. Warring, (336/$13.95/$21.95).

Reader Service No. 127.

DDJ

Clinic

(Continued from page 10.)

our CP/M 2 system had. So it was a disappointment to find that an assembly that ran in 25 seconds under CP/M 2 consumed 23 seconds under CP/M 3. A ten percent improvement didn't seem like much of a return on our investment in RAM cards. We set out to find out what the CP/M 3 BDOS was doing with all those buffers.

Inter-Bank Peeping

We wrote an experimental tool called SHOWBCB. It finds and dumps the two chains of the BCBs to the console. This is not easy, because the Disk Parameter Header, the chain anchors, and all the BCBs are in bank zero, while the program runs in bank one. It is made possible by the XMOVE and MOVE BIOS functions. MOVE is normally an intra-bank block copy function. If XMOVE is called first to inform MOVE of the source and target bank numbers, the next call on MOVE will transfer data from a source address in one bank to a target address in another. With them, our program could get copies of the control blocks into bank one so it could display them.

One problem slowed us down for a few hours. The program was written to play by the rules of CP/M 3, making its BIOS calls by way of BDOS function 50. It turns out, however, that the BDOS uses MOVE (and XMOVE?) in the course of executing function 50. So by the time we reached the BIOS for the MOVE function, the BDOS had already been there and thus had used up the parameters we had given to XMOVE. What we meant for an inter-bank move was thus magically transformed into an intra-bank move. We reverted to calling MOVE and XMOVE directly. This is possible provided that the BIOS routines and the calling program's stack are all in non-banked storage.

The Directory BCB Chain

Examine Figure 1 (page 91). It shows the BCB chain immediately after a cold start. The only disk I/O that has been done to this point is the load of CCP.COM and its load of SHOWBCB.COM. The lengthy display of the data BCBs has been trimmed to save space.

The first byte of a BCB is the number of the drive for which it holds a sector, and FFh indicates an unused buffer. Only one BCB is active in each chain. In the directory chain you can see that four BCBs have been used, but three of them have since been discarded. Old BCBs for track 2 sectors 1, 2, and 4 can be seen. Presumably sector 3 was read as well, but its BCB doesn't show.

Track 2 sector 4 was read again, and its BCB is still active. This BCB probably represents the I/O done to open SHOWBCB.COM. Thanks to its directory hashing table, the BDOS didn't have to scan the directory for SHOWBCB; it was able to read the one sector that contained its entry. However, it did have to read that sector because the residual BCB for the same sector had been discarded.

This seems reasonable on reflection. It's only by reading a directory sector and testing its contents against the hash table that the BDOS can detect a change of disks. If it didn't make that test, it would be as liable to trash a directory as MSDOS is. Extra directory I/O is a reasonable price to pay to avoid that danger to data integrity, so the fact that the BDOS seems to toss away its active directory BCBs shouldn't disturb us too much.

The Data BCB Chain

Now look at the chain of data BCBs. Two have been used, but only one remains active. The discarded one probably represents the last sector of CCP.COM; track 4 sector 6 is about the right location for it on this disk. Of course, CCP.COM occupies 3.5 one-kilobyte sectors. Why does only one BCB appear? Probably because the BIOS's cold-start code uses a multi-record read to load the file. When the BDOS knows it will be transferring eight or more 128-byte units from adjacent disk locations to adjacent storage locations, it probably skips allocating a deblocking buffer and reads the sector directly into storage. The last sector of the file is not all used, so a BCB was allocated to hold it while the last few records were pulled out of it. The one active BCB probably represents the only sector of SHOWBCB.COM.

Examine the addresses in the "bcb" and "link" columns; you can see how the BDOS's LRU algorithm works. BEC3 is the last BCB assembled in the BIOS. When the system is loaded, it is at the end of the chain. Whenever the BDOS needs a BCB, it removes the last one from the end of the chain and inserts it at the head of the chain. In Figure 1, you can see that the BCB at BEC3 was moved from tail to head, and the same was done to the BCB that originally preceded it, BEB4. The original chain head, BAA9, has aged two positions. We can tell how many BCBs the BDOS has used by counting the BCBs that precede BAA9. This remains true only for 71 uses, when the chain wraps.

BCB Use for Input

Our next experiment was to run an assembly that did input but no output ("rmac showbcb $1brzszpz"). The assembler had to read a total of 29 sectors: ten sectors of SHOWBCB.ASM (twice each), and nine sectors in three library files. Figure 2 (page 91) shows the data BCBs afterward. The only one still active, BE4B, represents the load of SHOWBCB.COM. The two oldest, BEB4 and BEC3, are left over from Figure 1. The six in between are the total result of loading RMAC, reading the source file twice, and reading three libraries. The BCB at BEA5 probably represents the last sector of RMAC (it, like CCP.COM earlier, would have been loaded with a multi-record read). That leaves five BCBs to account for all 29 sectors of data.

Two things stand out in Figure 2. First, the BDOS does not use a new BCB for every new sector of data it reads. If it did, Figure 2 would be 24 BCBs longer than it is. Perhaps the BDOS assigns only one BCB to each open file (three libraries plus the source file twice equals five). Perhaps it reuses a BCB whenever its current sector is completely used up and the next sector is for the next adjacent record of the file. The second scheme would produce the same numbers in this case of all-sequential I/O, but would allow for the accumulation of more BCBs to a direct-access file.

Either way, the result is that for sequential input, the BDOS effectively assigns a single BCB to each open file. Sequential data does not accumulate in the BCB chain. When a file is used twice in succession (as is the source file in an assembly), it will be read twice from disk. The only effect of the BCB chain is to eliminate the second read of the *last* sector. Now we can see why an assembly runs at essentially the same speed in CP/M 3 and CP/M 2.

The second thing that stands out in Figure 2 is that what little data was accumulated has since been discarded. The data residual from RMAC's execution was tossed out sometime between the end of RMAC and the start of SHOWBCB. The culprit had to be the CCP – it was the only program to run in that interval. We ran CCP.COM under SID to see what it would do.

What it did was issue BDOS request 98, Free Blocks. This BDOS function is supposed to release any disk blocks that were allocated to files that were never closed. It recovers disk space that might otherwise remain inaccessible until the disk was logged in again (possibly a long time in CP/M 3, since disks are only logged in when control-C is pressed). Exactly that problem occurs under MSDOS when the IBM Pascal compiler aborts, but under MSDOS you have to run CHKDSK to recover the space.

BDOS function 98 has a laudable goal, but it seems to have an unfortunate side effect. It not only cleans up the allocation vector, it also puts FFh into every active BCB! We proved that by killing the

call on function 98 in CCP.COM (it's at location 458h) and re-running the experiments. With the modified CCP, the BCB chain after the assembly had eight active entries instead of eight dead ones and one live. However, this had little effect on performance, since the active entries represented only the last sectors of their respective files.

BCB Use for Output

We checked BCB use following an assembly that produced PRN, REL, and SYM files. We expected to see BCBs for at least the last sector of each output file. We did not. We don't know what space the BDOS is using to block the physical sectors of output files, but it isn't space taken from the BCB chain.

To make sure that we weren't seeing the result of some clever use of multi-record writes in RMAC, we ran a test using the Magic Wand PRINT program with disk output. This program has abysmal disk I/O logic; it alternates reading and writing of 128-byte units. Under CP/M 2, with a BIOS that had only a single sector buffer, PRINT caused a seek, read, seek, read, write sequence for every 128 bytes of throughput. Under our 2.2 BIOS with separate read and write sector buffers, it ran better. In fact, it ran better than under CP/M 3, where PRINT takes about 10% *longer* than under our improved 2.2 BIOS! And its output didn't leave any evidence in the BCB chain.

Here again, it seems that the CP/M 3 BDOS is throwing away an opportunity to speed up processing. All output goes straight to disk; if it then becomes input, it has to be read back again. The file that one program writes, another program is going to read. The editor's output is input to RMAC; RMAC's REL file is input to LINK; and its PRN file is input to TYPE or PIP. It's all very well to speed up direct access, but we should keep in mind that (1) the majority of CP/M commands read files sequentially, and (2) commands are used in sequence, with the output of one feeding the input of the next. That situation is exactly right for the LRU algorithm, but the BDOS isn't capitalizing on it.

Well, almost exactly right. If the BDOS buffered all the sectors it read, and also buffered all the sectors it wrote, some problems would arise. If an output file was larger than the total buffer pool, its last few sectors would displace its first few. The next command's input of the first few sectors of the file would kick out the next few, and so on. A ripple would run down the BCB chain, causing each saved sector to be discarded just a few sectors before it was needed as input.

But the BDOS does *not* buffer all the sectors it reads. If it kept the present logic for sequential reading but buffered all output sectors, then the ripple would never develop. One BCB would be used to read the leading sectors that had been displaced by later output. Eventually the input requests would work around to the sectors that had not been displaced. From there to the end of the file, the second command would operate at RAM speeds.

Summary

A sophisticated algorithm for sector buffering under CP/M would make special provision for COM files, encouraging the retention of commands in storage. It would probably use separate chains of

```
Directory BCB chain...

 bcb drive record wflg "00" track sector buffad bank link *

BA6D   00  000008  00   08   0002   0004  4000   00  BA7C
BA7C   FF  000008  00   08   0002   0004  4400   00  BA8B
BA8B   FF  000000  00   00   0002   0001  4800   00  BA9A
BA9A   FF  000018  00   07   0002   0002  4C00   00  BA31 *
BA31   FF  000000  00   00   0000   0000  3000   00  BA40
BA40   FF  000000  00   00   0000   0000  3400   00  BA4F
BA4F   FF  000000  00   00   0000   0000  3800   00  BA5E
BA5E   FF  000000  00   00   0000   0000  3C00   00  0000 #

Data BCB chain...

 bcb drive record wflg "00" track sector buffad bank link *

BEB4   00  000BF0  00   04   0031   0003  7800   03  BEC3
BEC3   FF  0000B8  00   00   0004   0006  7C00   03  BAA9 *
BAA9   FF  000000  00   00   0000   0000  5000   00  BAB8
BAB8   FF  000000  00   00   0000   0000  5400   00
BAC7   FF  000000  00   00   0000   0000
BAD6   FF  000000  00   00                         03  BE69
BAE5   FF  000000                        0000   6400  03  BE78
BAF4                              0000   0000   6800  03  BE87
                          00   00   0000   0000   6C00  03  BE96
                    FF  000000 00   00   0000   0000   7000  03  BEA5
BEA5   FF  000000  00   00   0000   0000  7400   03  0000 #
```

Figure 1.

```
Data BCB chain...

 bcb drive record wflg "00" track sector buffad bank link *

BE4B   00  000BF0  00   04   0031   0003  5C00   03  BE69 *
BE69   FF  000C68  00   01   0033   0008  6400   03  BE5A *
BE5A   FF  000C60  00   07   0033   0005  6000   03  BE78 *
BE78   FF  000D08  00   04   0036   0004  6800   03  BE87
BE87   FF  000CE0  00   05   0035   0005  6C00   03  BE96
BE96   FF  000D30  00   03   0036   0003  7000   03  BEA5
BEA5   FF  0006E8  00   01   001D   0008  7400   03  BEB4
BEB4   FF  000BF0  00   04   0031   0003  7800   03  BEC3
BEC3   FF  0000B8  00   00   0004   0006  7C00   03  BAA9 *
BAA9   FF  000000  00   00   0000   0000  5000   00  BAB8
BAB8   FF  000000  00   00   0000   0000  5400   00
BAC7   FF  000000  00   00   0000   0000
BAD6   FF  000000  00   00
BAE5   FF  000000
BAF4
```

Figure 2.

BCBs for input and output, with some kind of heuristic to shuffle buffers from one chain to the other on the fly. In fact, CP/M 3 has enough resources to make it quite interesting as a case study in performance optimization. It could be the grist for a Ph.D. thesis in Comp. Sci., in fact.

But the existing algorithm is not sophisticated and, while it may be effective for direct access, it is definitely not optimal for the kind of sequential access that is the bread and butter of any CP/M system. Until it is improved (CP/M 3.2?), we recommend that you not spend a lot of money on RAM cards when moving to CP/M Plus. One 48Kb system bank should provide as many sector buffers as the BDOS can use effectively.

DDJ

Dr. Dobb's Journal

RED

RED's most important new feature is a set of buffer routines which remove all restrictions on the size of files which may be edited. The differences between RED and ED2 are all under the covers . . .

Dr. Dobb's Journal
For Users of Small Computer Systems

July 1983 Volume 8, Issue 7

Publisher – Clifford West
Editor – Reynold Wiggins
Managing Editor – Craig LaGrow
Contributing Editors –
Robert Blum, Dave Caulkins, Dave Cortesi, Ray Duncan, Michael Wiesenberg
Marketing Director – Beatrice Blatteis
Advertising Director – Carl Landau
Advertising Sales – Doug Millison
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Terri Marty, Linda Marohn
Production Manager – Barbara Ruzgerian
Production Assistants –
Alice Hinton, Ann West
Marianne Tryens
Typesetter – Paula Fairchild

Copyright © 1983 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: DeWitt S. Brown, Burks A. Smith, Robert M. Connors, Robert C. Luckey, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Thomas Davis, Jan Steinman, G. Hunter, Ronald E. Johnson, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III, Infoworld, Stan Veit, Western Material Control, S.P. Kennedy, *Ed Moran*, W.D. Rausch, John and Janith Hatch. **Lifetime Subsriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

CONTENTS

ARTICLES

12 Augusta, Part IV – The Augusta Compiler (continued)
by Edward Mitchell
Concluding the four-part series on his tiny language, author Mitchell describes the internal operation of the Augusta compiler. Included are details of the lexical analyzer, the symbol table, code generation, and backpatching techniques, as well as some personal comments on Ada, Microsoft BASIC, and overall development of the compiler.

34 RED – A Better Screen Editor, Part I
by Edward K. Ream
In our January 1982 issue the author described ED2, his public-domain text editor. Extensive improvements produced RED, including new buffer routines which remove all restrictions on the size of files which may be edited. This month the author discusses those virtual memory routines, and a code-saving method of error recovery. The rest of RED will be discussed next month.

65 Anatomy of a Digital Vector and Curve Generator
by Richard Shiffman
In past issues we have presented various methods for drawing lines and circles. This article presents a slightly different angle, providing BASIC implementations of both a vector and circle generator.

70 Fast Matrix Operations in Forth, Part II
by Steven A. Ruzinsky
In the June issue, Part I of this series presented a matrix-defining word and a few accessories. This installment introduces programs for performing matrix operations, as well as some more advanced accessories.

DEPARTMENTS

6 Letters

6 Editorial

80 Dr. Dobb's Clinic
Two Gets You Ten, Cheap Thrills Dept., and the AGGHHH! Program

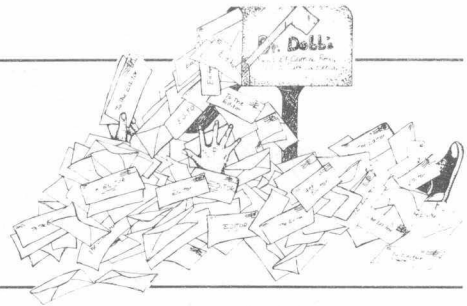
86 CP/M Exchange
MBOOT Revisited, CP/M Plus Feedback, a bit on CCP replacements, and 64K and Beyond

92 16-Bit Software Toolbox
MS-DOS Rebuttal, 68000 Tools, and Sizing Memory on the IBM PC

100 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

LETTERS



Dr. Dobb's Journal *welcomes letters to the editor as a forum for ideas, innovations, irascibility and even idiosyncrasies. Some letters may be edited for clarity and brevity. The Doctor likes hearing from you – keep on writing!*

Catching Bugs in Small-C

Dear *DDJ,*

May I again borrow your pages to point out a few bugs in Small-C V2.0, and offer the code (see page 9) for the missing selection operator?

There are bugs in 'peephole' and 'pstr' which are very similar. 'Peephole' – the section of code (TAB defined) which substitutes pop-push sequences for fetches of local variables – should be changed to test for a trailing ' \tXCHG;; ' rather than simply ' XCHG;; ', with the obvious change to the offset in the staging buffer. If this is not done, the NULL (which marks the end of the current output string) might be skipped. In 'pstr', change '++lptr' to 'gch()' to force an update of 'ch'; otherwise the NULL marking the end of the current input line will be skipped if the end of the line in a multi-line expression is a character constant. This results in the trailing end of a longer previous line's being appended to the current line.

I am indebted to Mr. M. Grundy for pointing out this second bug.

A further bug lies in wait for the unwary in the function 'decIloc'. As the code stands, one must declare automatic pointers as '*ptr' rather than 'ptr[ ]' since no space is allocated on the stack for this second form. The fix is to change the 'else' clause of the test 'if (k=needsub())' to a compound statement by adding 'k = BPW; ' and the enclosing braces.

In the evaluation of expressions, the comma operator is not fully implemented. The expression

```
i = ( j = k , 1 )
```

should result in the j being assigned the value of k, and i the value of 1. What actually happens is that the value of k is assigned to both i and j, and the 'missing token' error is generated after k. For a correction (using the features of the V2.0 compiler) locate the function 'primary' (in CC32.C) and insert the marked lines:

```
    if ( match ("(") )
        {
#       do
                k = heir1 ( 1val );
#           while ( match (",") );
        needtoken (")");
        return k;
        }
```

There is a further minor bug – more in terms of style error – which users of Small-C may become prone to. The old forms of the assignment operators are not tested for:

'=* =/ =% =+ =- =<< =>> =^ =| =&'

Most of these will cause no problems on on transfer to a full system, but three are obviously ambiguous (i.e., '=* =& =-'). The UNIX compiler generates warnings for these forms; and, for '=+', it parses all four as assignment operators. The Small-C compiler requires some editing to check for these.

I offer the code for that most useful of 'C' operators – the selection operator – with one caveat: the code as presented conforms to normal practice on if-else binding, albeit an implied else in this case. The result does not parse in the same way as the C compiler on the UNIX system. Whereas I may write:

```
x ? y ? xandy : xandnoty : notx
```

the UNIX compiler requires explicit bracketing thus:

```
x ? ( y ? xandy : xandnoty ) : notx
```

The 'bible' (Kernighan & Ritchie) does not resolve which of these two is the correct implementation. I believe my approach conforms better to the rest of the language with the re-entrancy at each

EDITORIAL

Since its inception *DDJ* has been associated with a remarkable group of people: that rather diverse group dedicated to the proliferation of public domain software. These people freely share their labors with the rest of the computing community, providing a pool of ideas which the community at large may use, develop, and refine. Ultimately this benefits even the consumer of commercial products. Indeed, over the years this resource has been a significant factor in the general improvement of software.

Their approach to software distribution and development is not without its problems, however. When one releases rights in a piece of work, he or she also relinquishes the corresponding ability to regulate how that work is changed or used. Lately, we seem to be hearing more about bad experiences in this respect. One instance may be found in our Letters column this month. Distribution of programs without credit to the author is just one more example. None of the problems are new, but they are not likely to become less prevalent as the use of computers and the competition in the software market increase.

The source of these problems lies in the very nature of the public domain. External restraints would undermine the benefits currently enjoyed, and result in increased time and cost for inforcement. The public domain offers few legal controls to authors, but also few restrictions. Some degree of frustration probably must be endured in the name of the cause, relying instead on individuals to exhibit a sense of mutual respect. Due to *DDJ*'s involvement here, we would like to invite discussion from readers on possible solutions to the inherent problems.

Keep in mind that it is the sense of community, as much as anything, that has kept public domain software growing over the years. Often the only compensation which these authors receive for their work is recognition from their peers. Let them know you appreciate their efforts. Give credit when you use their material. Better still, write to them with your comments and examples of what you have done; perhaps ask them for feedback. Allow them the satisfaction of seeing their influence.

Reynold Wiggins

level of precedence, which can be seen in the code for the Small-C compiler. Perhaps the good Doctor could obtain a ruling? For compatibility substitute 'heir3' for 'heir2' in the lines marked with a '#' on the enclosed listing.

Yours sincerely,
Andrew Macpherson
29 Tisbury Road, Apt. 7
HOVE, E. Sussex
England BN3 3BJ.

In Defense of Ada

Dear Editor,

Being the author of Supersoft Ada, I was interested in the article by Ed Mitchell on his "Ada-oid" Augusta. I would like to add a few of my own comments to those already submitted.

On page 34 of *DDJ* No. 77, Ed lists results of the benchmark program found in an earlier *Byte* magazine. The results, for my Ada at least, are incorrect. The latest version of the compiler executes the program in 37 seconds, with a compile/load time of 41 seconds. The *Byte* article states that the latest available versions were used for the test, but I have never been able to get the benchmark to run that slow in any version. This is, of course, not Ed Mitchell's fault.

I have noted several letters appearing in *DDJ* and other publications with dismay; Ada does not seem to be a popular language among the microcomputer community. Considering Pascal's wide acceptance, and Ada's similarity to Pascal (which is not a coincidence), I am at a loss to explain the evident lack of support for the new language. I for one would like to come to its defense (no pun intended).

First, let me point out that I am not an Ada fanatic. I enjoy programming in Ada as much as I do in any language, including Pascal, C, and yes, even BASIC. Ada has much to offer a programmer for some applications. I wouldn't try to write a text-oriented editor in it, it has lousy string handling capabilities; but then I wouldn't try to write a tax program in LISP, or an adventure game in COBOL. The DoD wanted a language to write programs for embedded systems such as guided missiles. The language they ended up with just so happens to be a good structured language that is also useful for general scientific programming. A programmer who is intelligent enough to be able to select the proper language for a given application, whether he likes programming in that language or not, is indeed a valuable asset.

The language has its faults. Operator overloading certainly won't make a program more readable or maintainable, the I/O stinks, and most people think that Ada tried to bite off more than it could chew. Nevertheless, one fact remains: the government is backing Ada 100% and you can bet that good Ada programmers will soon be in demand.

Ada does have its strong points. It is a strongly typed language, a characteristic well received by programmers as shown by Pascal. It has better structure than Pascal, solving the "dangling-else" problem. Resulting code is much easier to read, and the flow of control easier to see in an Ada program. Ada has a powerful set of basic statements, and its packaging facilities and multi-tasking capabilities are relatively new concepts being introduced in a single language. Corporations such as Honeywell, SofTech, and Western Digital have invested significant effort in their respective Ada projects.

The discussion of Ada might not have been even brought up if Ed had called the language "STRUCT-1," or "NEWPAS," or something like that. I am of course referring to the DoD's Boolean logic concerning Ada compilers. Augusta bears less of a resemblance to Ada than one might think. The implementation looks more like a language oriented toward graduate courses in structured language design: a "compiler-writer's language" so to speak. I am not saying I approve of the DoD's position; I think that, if they approved subsets, the acceptance of Ada would be more widespread. Let me point out that I stated SUBSETS, not "variants on the Ada language with extensions." Also, I am not trying to put down Ed's work. I know the difficulties in implementing a structured language and I applaud his success.

Count me as one of the supporters of the Ada language. I welcome any comments or suggestions sent to my address;

Listing – Selection Operator

```
/*
** Selection operator, new function 82-12-16, A.Macpherson.
**
** selector ? exprl : expr2
**
*/
heir2 ( lval )
int     lval [];
{
int     k;
        k = plungel ( heir3, lval );
        if ( match ("?") )
                {
                int       falselab, endlab, val;
                char      reg2used, selconst, explconst,
                          sel2;
                falselab = getlabel ();
                reg2used = 0;
                if (k) rvalue ( lval );

                if (selconst = lval [ 3 ] )   /* selector is a
                                                 constant */
                          {
                          if(sel2 = !lval[4]) jump(falselab);
                          }
                else      {
                          testjump (falselab);
                          reg2used = lval [ 5 ];
                          }
#               dropout ( plungel (heir2, lval), jump,
                          endlab = getlabel (), lval );

                if (explconst = lval[3])      val = lval[4];
                else                    reg2used |= lval[5];

                postlabel ( falselab );
                needtoken (":");
#               if ( plungel (heir2, lval))   rvalue( lval );
                if ( lval [3])
                          {
                          const (lval [4]);
                          if (selconst && sel2)
                                    val = lval[4];
                          }
                else reg2used |= lval[5];
                lval [ 5 ] = reg2used;
                lval [ 4 ] = val;
                lval [ 3 ] = selconst && ((!sel2 && explconst)
                                    || ( sel2 && lval[3]));
                postlabel ( endlab );
                return 0;
                }
        return k;
}
```

or you may leave CompuServe correspondence at 71705,2037. Thank you.

Sincerely,
David C. Norris
Maranatha Software Systems
500 Catalina Road, No. 305
Cocoa Beach, FL 32931

Public Domain Responsibilities

To *DDJ* Readers and CP/M Users,

Over the years that *DDJ* has been serving us, I have contributed one article (a disassembler in issue no. 27), a review of the PC-1 pocket computer (in issue no. 52) and a few letters. However, this contribution is a bit different. The bottom line of this letter is a warning!

I am the author of a set of programs which are in widespread public domain distribution (at least three continents). They are called SQ.COM, USQ.COM, and FLS.COM. They run on CP/M systems and serve to squeeze files to smaller sizes in preparation for transmission via modem or distribution via diskette.

The latest versions of these programs are 1.5, 1.5 and 1.1, respectively. I released these programs to the public domain, *except* for financial gain, and included full source code (in C language) and extensive documentation. This software is now available via the C Users group, which has advertised in *DDJ*.

Having had reasonable experiences with previous public offerings, I gave my name and address in the documentation and in the sign-on message of each program. Therein lies the story!

In recent months I have had a couple of dozen phone calls claiming that these programs have destroyed various innocent files. But when I ask for details, I find that the programs in question have version numbers higher than any I have ever issued! Not only that, but many different version numbers, as high as 1.9, have been mentioned!

I expect that whoever produced each version of these programs was as innocent as I was in August of 1981 when I issued the above-mentioned programs. *But* the fact remains that one or more persons has taken it upon themselves to propagate modified versions of my software without taking the responsibility of adding their name and address to the documentation and sign-on messages.

It is an unfortunate fact that C and probably other high-level languages can crash impolitely when the CP/M TPA is too small for the program and its data space, including dynamically allocated local variables. (These programs *do* use recursive algorithms.)

I have heard, without confirmation, that vast amounts of code have been added to some versions of these programs to add wild card file name processing. Perhaps this is the problem. I provided that feature as a separate program, FLS.COM, precisely for that reason. It is also possible that full and careful use of the recompilation options available in BDS-C has been abused, resulting in too much code or improperly linked code. Or maybe there have simply been some accidents in copying or modem transmission.

Whatever the case, I recommend that if you modify software which you have not written, then try to consult with the original author before publicly distributing such software. If that is not possible, at least accept the responsibility of being the first line of complaint handling by adding (not replacing) your name to the documentation and sign-on message.

If anyone has a file which the above-mentioned versions of my programs cannot squeeze and then reproduce, please contact me. My computers can read and write the following diskette formats: Micropolis Mod II, Lobo Max-80 5" 40 track/side (SS/DS, SD/DD) formats, Osborne Single Density, Omikron 5" single density, Xerox 5" single density. Except for the Micropolis, all of the above are handled by my Lobo Max-80 computer.

Do you really want more public domain source code software? If so, and if you know the source of these modifications, please help stamp out the bad versions (copies?) of this software. Unfortunately, I don't know exactly which versions are bad because not one single person has ever sent me a courtesy copy of any modifications!

To those bewildered innocent victims, the unsqueezing process contains a validity check based on the original file. If you can produce a file (not just a console display) by running the USQ.COM program, then the resulting file should be an exact replica of the original.

I also ask purchasers of communications packages who I have authorized to distribute this software not to assume that I am responsible for the communications packages. It seems that reproduction of my documentation files makes my name and address more prominent than those of the commercial distributers!

Sincerely,
Richard Greenlaw
251 Colony Ct.
Gahanna, Ohio 43230

Moaning & Groaning About MS-DOS

Dear Dr. Dobb,

I have to take issue with Jim Howell's comments regarding the MS-DOS batch facility (*DDJ* No. 79, 16-Bit Software Toolbox). But more of that later.

Over the past few months, I have been improving various CP/M-80 v.2.2 utilities to be the standards I have been used to on ISIS systems. One of these jobs involved upgrading SUBMIT to handle multiple batch files and nested batch files. In this, various issues of *DDJ* have proven themselves more than useful, for which many thanks. I have one remaining task to perform which I think should be relatively simple but may interest other readers. I wish to halt the execution of a batch running under XSUB after the command line has been read but before it is executed. I then wish to perform an acceptance of the command by hitting return or to cancel it with a control-x. Probably the easiest solution would be to write a small program to do this, but I am intrigued with the idea of modifying XSUB.

Now, turning to the point of contention with Mr. Howell, the above task was relatively easy to perform using DDT. (In fact I disassembled SUBMIT and XSUB and proceeded that way.) When I came to try the same job on the MS-DOS batch facility, it has so far proved impossible as the "SUBMIT-equivalent" is buried somewhere in the DOS code. Further, there seems to be no "XSUB-equivalent" allowing called programs to accept input from the batch rather than the keyboard. In these regards, I find the MS-DOS batch facility to be much inferior to the DRI product.

A final moan and groan about MS-DOS, or at least about IBM's packaging of Microsoft's software: I purchased the IBM MACRO ASSEMBLER only to find later that I have to also purchase a high-level language compiler to get a library manager utility. I wonder if the people who package these products ever use them themselves. Also, the Microsoft brochure on MS-DOS mentions a TRANS utility for translating MACRO-80 source code into MACRO-86 source. Has anybody out there seen this utility -- it seems to be impossible to obtain.

Lastly, much praise and thanks to you and all the individuals for all the useful articles on Small-C.

Sincerely,
Bill Potter
425 Fair Drive, No. 205
Costa Mesa, CA 92626

Still Searching for The Right Byte

Gentlemen:

I have just finished reading the excellent article in *DDJ*, No. 78, regarding the disk parameter block by Gene Head in "CP/M Exchange." It came just in time to prevent my going to the insane asylum.

(Continued on page 105)

AUGUSTA, Part IV

The Augusta Compiler (continued)

Edward Mitchell began this series in January of this year. Due to the size of Ada, Augusta was intended to be a (very) tiny language in the spirit of languages such as Tiny BASIC, Tiny Pascal and Small-C. During the ensuing months he has presented an in-depth look at the structure and implementation of Augusta, intending to illustrate how to build such a language. Regardless of philosophical differences one may have with the author concerning language implementations, the discussion is bound to have been stimulating.

Part IV concludes the series by describing the internal operation of the Augusta compiler. Also in this discussion are details of the lexical analyzer, symbol table, code generation, and backpatching techniques, plus some personal comments on Ada, Microsoft BASIC, and the overall development of the compiler.

General Features

Microsoft BASIC, like most BASICs, lacks constant symbol declarations. Because of that, a large number of BASIC "variables" are used in the compiler as "constants." For example, all of the p-codes generated by Augusta are labelled with mnemonic variable names. Rather than generate "1" for the LDCI opcode, it is much clearer to refer to PLDCI which always has the value 1.

Similarly, whenever the lexical analyzer sees a keyword, it returns an integer that uniquely identifies that keyword. As described in the next section, each call to the scanner sets variable T equal to an integer that represents the token. For example, BEGIN sets T=KBEGIN, where KBEGIN is an integer constant. Thereafter, if the parser is expecting a BEGIN statement, it can test by

IF T=KBEGIN THEN . . .

Comparing integers is much faster than comparing strings (e.g., comparing T$ = "BEGIN" is slow). Punctuation symbols, like "(", "+", and ";" are also represented as integers. Lines 1130 through 1210 of the compiler (see Listing Two of "Augusta, Part III," page 20, *DDJ* No. 79, May 1983) initialize these "constant" variables. Note that all variables inside the compiler are integers unless explicitly labelled as some other type.

The variable names are mnemonics for the tokens they represent. SQUOTE is a single quote, C is a constant, LP is a left parenthesis, and RP a right parenthesis. MUL, DIV, ADD, and SUBT correspond to the arithmetic operators "*", "/", "+", and "-". LES, LEQ, EQ and so on refer to the relational operators "<", "<=", and "=". ID is an identifier and CH a character constant. SC is a string constant. The rest of the tokens should be apparent from their "constant" names.

The compiler is organized into numerous subroutines, each identified by a comment on the first line of the routine. Table 1 (page 20) is a cross reference of the major subroutines and their location and function. Table 2 (page 21) is a dictionary of major variables in the compiler. Many subroutines take their names directly from the boxes shown in the syntax diagrams in Part I (*DDJ* No. 75, January 1983). For example, the routine labelled SEQOFSTMTS corresponds to the block in the syntax diagram labelled SEQ OF STMTS (see pages 24-25 of the January issue).

Compiler Initialization

All of the keywords and predefined procedure names are kept in a data file titled "KEYWORDS.TXT" (see Listing 2, page 32). When run, the compiler opens the file and reads in the identifiers.

Keywords are stored in a fast look-up table, array KEY$(), hashed by the keyword's first letter. The first letter ("A"=1, "B"=2, "C"=3, etc.) indexes an array MAP(i). MAP(i) is essentially a hash function, pointing directly to the set of keywords beginning with that letter. MAP(1) is the index in KEY$(j) of the keywords beginning with "A". Map (2) points to the keywords beginning with "B" and so on. The lexical analyzer uses the hash table to quickly identify keyword symbols.

The Lexical Analyzer

When the parser needs a token, it calls the subroutine at line 1400 labelled "GETSYM." GETSYM examines the character that the scanner last read (in CH$) and branches to the appropriate state. For example, if the character is an "A", "B", "C", . . . , or "Z", then the analyzer jumps to line 1460 and reads an identifier. If the character is a digit, then it jumps to line 1490 where it scans for an integer.

The lexical analyzer is initialized by calling GETLINE, which inputs a line of the Augusta source program into BUF$. GETLINE then tacks an "end-of-line" marker onto BUF$ and skips over any leading blanks. It then sets CH$ equal to the first "real" character in the line and sets B to be the index in BUF$ of the next character.

GETSYM looks at the character in CH$ and processes it accordingly. As GETSYM needs additional characters it calls GETCH, which sets CH$ to the character at position B and increments B. When GETSYM detects the "end-of-line" marker it calls GETLINE to read in the next line from the source file.

GETSYM sets variable T equal to the integer representation of the token. Where needed, S$ holds the actual token string. TT$ is set to CHR$(T) and is used in conjunction with the BASIC INSTR function to quickly check if T is a member of a set of tokens. For example, RELOPS$ is a string containing all of the token values for relations (e.g., "=", "<=", etc.). The parser can quickly determine that T is a relational operator by checking to see that INSTR(RELOPS$, TT$) is greater than zero.

The Symbol Table

As the source program is parsed, the compiler needs to keep track of the variables and procedure names that have been defined. Several techniques can be used to manage a symbol table. Binary trees, hashing, or ordered tables come to mind. But it turns out, at least in interpretive BASIC, that the better methods are quite slow.

Searching a string or strings in a sequential manner using the INSTR function is much faster than a binary search written in BASIC. This observation applies to fairly large tables as well. Consequently, Augusta's symbol table uses a stack-like mechanism which is then searched sequen-

by Edward Mitchell

Edward Mitchell, P.O. Box 390145, Mountain View, California 94039.

This series Copyright © 1983 by Edward Mitchell. The Augusta Compiler is Copyright © 1983 by Computer Linguistics. "Augusta" is a trademark of Computer Linguistics. "Ada" is a registered trademark of the U.S. Department of Defense.

tially by scanning it with INSTR.

As symbols are encountered while parsing the program they are placed onto the top of the stack. As shown in Figure 1 (page 16), the symbols for procedure P2 are placed on the stack so that the most recently defined symbol is on top. Searches for a particular identifier always begin at the top of the stack. Consequently, the most recently defined symbols, such as local identifiers, are encountered first. And since most searches are for local variables, the search terminates fairly quickly.

Rather than using an actual stack, a string variable is used to hold the symbol table. The first symbol encountered is placed in S$. The second symbol is *inserted* into the front of S$ as,

```
S$ = ID$ + S$
```

Therefore, if two variables are defined in a program as,

```
I    : INTEGER;
J    : INTEGER;
```

then S$ is "J I", with J before I in the table. The table is then searched left to right by executing,

```
INSTR( S$, ID$ )
```

where ID$ is an identifier to look up. This means that if a second variable I is defined locally to a subsequent procedure it will get placed *before* the other I in S$. Then, when INSTR is used to search S$, the most recent definition of I is found first.

There is a problem with this scheme: in Microsoft BASIC, strings cannot be longer than 255 bytes. The solution is to maintain an array of strings S$(1), S$(2), S$(3), ... When S$(1) is about to exceed 255 bytes in length, S$(2) gets used. When S$(2) fills up, S$(3) is used. The strings are then searched by first looking at S$(3), then S$(2) and finally S$(1). The subroutines at lines 3850 and 3890 (see Listing 1, page 22) add symbols to the stack and perform searches, respectively. Figure 2 (page 17) shows the detailed layout of a single 17-byte symbol table entry.

Because the symbol table is a stack, the compiler cannot do type checking of procedure parameters. After the procedure is compiled, the local and parameter variables are removed from the stack, leaving only the procedure's name. Doing this substantially reduces the symbol table's memory requirements.

When the compiler parses a procedure call like,

```
SEARCH ( ELEMENT, @FOUND );
```

the symbol table no longer contains the list of parameters originally defined for SEARCH. Because of that, the compiler cannot detect type mismatches between the actual parameters and the parameter variables. Nor can it detect whether the correct number of parameters was used. In addition, the special symbol "@" is needed to tell the compiler when to pass a parameter by reference.

The Parser

Recursive-descent parsers, like the one used in Augusta, were described in detail in Part III. Since Augusta is written in BASIC, recursive-descent parsing requires extra code to handle recursive subroutine calls and the saving of data onto an internal stack.

For example, the CASE statement may itself contain other CASE statements as part of the code executed for the various conditions. When that happens, the same subroutine is entered twice – once to begin parsing the outer CASE statement, and the second time to parse the inner CASE statement. All variables that are unique to each CASE must be preserved by pushing them onto an internal stack. Upon return from processing the second (or third, or fourth, . . .) CASE, the variables are restored by popping their old values from the stack. Other than that, the parser works as described last month.

The compiler uses an array S() as a stack and variable SP as the stack pointer. A single integer X is pushed onto the stack by calling the PUSH subroutine at line 4280. S(SP) is set equal to X, and SP is incremented by one. Values are popped by calling the POP subroutine at line 4300. The overall size of the stack is determined by the DIM S() statement in line 1790.

Parsing expressions requires the compiler to keep track of the "type" of the values it is dealing with. When trying to add two values together, it must verify that both values are really numbers, and not strings or characters. The array TY() and the "Type Stack Pointer" TSP are used to perform type checking.

The code for parsing expressions appears in the subroutines labelled EXPR (line 3100), RELATION (line 3190), SE (line 3290), TERM (line 3350), and PRIMARY (line 3610). Compare these routines to the similarly named sections of the syntax diagrams shown in Part I.

Within PRIMARY, p-codes are generated to load the values of variables and constants onto the run-time stack. When the compiler sees a constant, such as the integer 517, it generates an instruction,

```
LDCI 517
```

Simultaneously, it places a type marker on the type stack inside the compiler. It pushes the type TINT (a "constant" variable that is mnemonic for "Type INTeger") on the stack TY().

Let's assume that the statement being parsed contains

```
I := 35 + 517;
```

Upon seeing the identifier I and determining that it is an integer variable, Augusta puts a TINT on the TY() stack. Next, at the constant 35, Augusta again puts TINT on the stack. The same occurs for 517. There are now three integer types on the type stack. Since addition can only be performed on integers, the compiler checks that the top two values on the type stack are both integers. Since they are, it removes both TINTs and generates the ADI pseudo code. But the result of ADI is an integer, so it puts one TINT back on the stack. Returning to process the "I :=" part of the statement, the compiler insures that the type of the expression is also an integer since it is being assigned to the integer variable I.

Type checking is also needed to handle relational operators. For example, when parsing the relation S1 = S2, the compiler needs to know if S1 and S2 are strings or integers since it generates a different p-code depending on the type of the relation (EQUSTR for strings, EQUI for integers).

Code Generation

Once the compiler has parsed a portion of a statement, it translates the statement into pseudo code. Looking to PRIMARY at line 3610, we can see how it generates the "load constant" instructions. At line 3630, if the token is a constant, the compiler places TINT on the type stack and calls the subroutine at line 3640.

Whenever GETSYM reads a constant, T is set equal to the C token and TN is set to the corresponding numeric value of the token (hence the mnemonic for "Token Number"). At line 3640 the compiler generates a p-code to load the value of TN onto the p-machine's stack.

The compiler must choose which of the various "load constant" p-codes is best to use. If TN = –1, then the compiler generates the SLDCN1 ("Short Load Constant Negative 1") opcode. If TN's value is from 0 to 15, it generates one of the single-byte short loads – SLDC0, SLDC1, etc. If the constant is greater than 15 but less than 256, it generates the SLDC "Short Load," followed by the single-byte constant to load. Otherwise, the compiler emits the LDCI "Load Constant Integer" opcode followed by the 2-byte constant.

The code is actually output by setting W equal to the opcode value (e.g., W = PLDCI) and calling the GENBYTE subroutine at line 3990. The variable CP ("Code Pointer") always contains the address where the next byte of code will be written. If necessary, an immediately following parameter is emitted by calling either GENBYTE or GENWORD as appropriate.

The actual mechanism by which the code bytes end up in the code file is somewhat complicated. First, let's look at a simplified version. The code file is always kept open as file #1 with a record size of 128 bytes. The current file buffer is accessed through a field statement that overlays the buffer. For example,

```
FIELD #1, 20 AS D$, 1 AS T$
```

places T$ at the twenty-first byte in the buffer. To output a byte, the compiler uses CP, the code address, to calculate which record in the code file will hold the code. Then it computes the byte offset within the record. By assigning,

```
LSET T$ = CHR$(W)
```

the byte is written to the code file buffer. CP is incremented by one to point to the next byte.

When the buffer fills up to 128 bytes it must be written to the code file and a new buffer prepared for additional code. Conceptually, you can think of the code generation working like this. But actually the compiler keeps three buffers of code in memory simultaneously. It does this because it sometimes has to "backpatch" code that has already been generated and it is much faster to access one of the buffers than to read a record from the disk.

Backpatching

When the compiler is producing code, it sometimes has to create forward jumps for which it does not yet know the destination. Part III, Listing 7 (page 29, *DDJ* No. 79, May 1983) showed the p-codes generated for an IF-THEN-ELSE statement. The statement following THEN ends with an unconditional jump (UJP p-code) to the first statement following the IF-THEN-ELSE. When the compiler produces the UJP instruction, it doesn't know where the jump should be to. Instead it must remember the location of the UJP instruction, and, after compiling the following statements, go back and fix them up.

In many instances, there are numerous UJP and FJP instructions that must be fixed up. A clever way to remember the location of the UJP instructions is to link them together into a list using the 2 bytes following each UJP. A variable LUJP is used in the compiler to hold the address of the most recent UJP instruction. The address field of the UJP instruction pointed to by LUJP points, in turn, to the previous UJP instruction. In that manner, all of the UJPs are linked together. An address of 0 signifies the end of the UJP list.

Once the statements have been compiled, Augusta goes back along the chain of UJP instructions and changes the links to be offset to the first statement following the ELSE.

Error Handling

Augusta's ability to handle syntax and semantic errors is limited. Missing semicolons cause the compiler to issue a warning and then to continue compiling the text. When other syntax errors are detected, the compiler prints an error code (see Tables 2 and 3 on page 21), the source line containing the error, and an asterisk "*" to the right of the last token read. It then displays,

```
REENTER +
```

You may then type in the remaining portion of the line in its *correct* form. For example, if your program contains,

```
IF A=B THEN PUTLINE ("A=B");
END;
```

Augusta detects the missing "IF" after the "END" and displays,

```
39 EXPECTED
ERROR 4
   END;
        *

REENTER + IF;
          —
```

At the "REENTER" prompt, the IF; is entered.

Semantic errors such as undeclared identifiers cause the compiler to print the above error message and then halt. Multiply defined identifiers are ignored – only the most recent definition is used.

Other Features

Augusta does not place a limit on the length of a procedure or a program. However, since the compiler is written in BASIC using 16-bit integer addresses, there is an implied limit of 30847 bytes of code for the entire program and 32767 bytes of either global or local data.

Personal Comments on Writing Augusta

Ada was designed for the U.S. Department of Defense as the result of studies conducted by the DoD during the mid to late seventies. The DoD realized that its software development and maintenance efforts were out of control. With programs written in numerous languages and dialects and then made to run on several hardware configurations, the DoD was faced with a nightmare.

A DoD-formed study group recommended that the DoD sponsor a design for a single, high-level language that would fulfill all the Department's data processing needs. The eventual result was Ada, named in honor of Augusta Ada Byron,

(a)
```
PROCEDURE P1 IS
   I,J : INTEGER;

   PROCEDURE P2 (X : INTEGER) IS
      Y,Z:STRING
   BEGIN
      NULL;
   END;

BEGIN
   NULL;
END;
```

(b) Contents of Symbol Table stack

```
Z     ← Top of stack
Y
X
P2
J
I
P1    ← Bottom of stack
```

Figure 1.

The symbol table uses a "stack" to keep track of identifiers. The most recently defined symbols always appear on the top of the stack. (a) shows a short program. (b) shows the symbol table when the compiler has reached the BEGIN in procedure P2.

the Countess of Lovelace and the daughter of the poet Lord Byron. She is credited as being the first computer programmer for the work she performed on Charles Babbage's mechanical computing engine during the mid 1800's.

The language that was produced was strongly influenced by a number of existing languages, including Pascal, Algol and SIMULA. While Ada contains little that is particularly novel, it does collect into a single language a number of state-of-the-art concepts, including "packages" for modular compilation, and multi-tasking.

Unfortunately, to satisfy everyone's needs, the language that was created became quite large. Just compare the size of the Ada draft standard with that of Pascal. The table of contents for the Ada standard is more than twice as large as that for Pascal. Further, Ada's complexity forces compilers to be large, complicated programs, ill-suited for small computers.

The DoD registered the name "Ada" as a trademark. Only compilers that implement the full Ada definition may be properly called "Ada." And there is no Ada subset because the DoD has refused to create a subset definition. (Most of the compilers currently advertised as "Ada" are, in fact, Ada subset compilers. They must, however, display a DoD-required notice that the compiler is currently incomplete but that it will be developed into full Ada and submitted to the government for validation at a later date.)

The lack of a well-defined Ada subset may hinder the language's growth. Full Ada will not run on low-cost, low-end systems, such as TI-99s, Commodore 64s, Ataris, or even Apples. And even development on large personal computer systems will be difficult. For example, IBM Pascal requires three floppy diskette changes just to compile and link a single program.

Ada is fine if you have access to a minicomputer or a "micro" with 512 Kbytes of memory and a hard disk. Realistically, Pascal's big push to popularity came from underneath, particularly with the advent of the UCSD Pascal system for personal computers. An Ada-subset definition would be ideal for introductory computing classes which, in turn, could provide a large pool of programmers skilled in the use of Ada.

(a)

Identifier	Type	Kind	PInfo	Const	ObjSz	Addr	Lex
8	1	1	1	2	1	2	1

(b)

Identifier 8-byte ASCII string, padded with spaces if necessary

Type The "type" of the identifier (e.g., INTEGER, STRING, etc.) encoded as a byte value:
0 = string
1 = integer
2 = character
3 = * not used *
4 = boolean

Kind "Kind" encoded as a byte value:
0 = constant
1 = array variable
2 = procedure
3 = function
4 = type definition
5 = scalar variable

PInfor "Parameter Information" indicates if the identifier is a procedure or function parameter, encoded as a byte value:
0 = Identifier is not a parameter
1 = IN parameter
2 = OUT parameter
If Kind = 4 Then
If PInfo = 0 Then Identifier is array type
Else Identifier is not an array

Const If Kind = 0 then this field holds the value of a constant; otherwise, it holds the size of an array.

ObjSz "Object Size" is the size in bytes of an element of the structure (e.g., an integer has an ObjSz of 2, a string has a size of 81, and an array of integers also has a size of 2).

Addr The address as an offset from the frame mark for the procedure in which the identifier is used (or the procedure number if the identifier is a procedure or a function).

Lex The lexical level of the identifier.

Figure 2.

Detailed layout of a symbol table entry. Each entry is 17 bytes long – 8 bytes to store the actual identifier and 9 bytes of information describing the identifier. (a) shows the entry format and (b) describes the fields in detail.

Designing Augusta

The choice of features in Augusta was influenced by the type of software that I write. That includes language processors, text editors, operating system routines, and an occasional game. I attempted to produce a true but powerful subset of the Ada language that would be useful for hobbyists on personal computer systems. Two minor extensions were made to Ada: the use of the "@" symbol for parameter passing, and UCSD Pascal style strings.

Once the compiler is rewritten in Augusta, these extensions will be removed and additional Ada features will be added, such as real arithmetic, "EXCEPTION" error handling, and package libraries for modular program development.

The subset that was chosen provides a language sufficiently powerful to be useful in a wide range of applications. Additional Ada constructions, such as packages, would be nice to have but are not needed for a boot-strap compiler. Instead, a group of "built-in" procedures and functions were created to support Ada-like terminal and file system access, string handling, and other functions. Once Augusta supports packages, these procedures can be easily moved into package definitions.

Why BASIC?

BASIC is a poor language for producing large programs. But there were several reasons that I chose – or, more exactly, was forced to choose – BASIC. Since I used an Osborne I for all program development, I had to fit everything – source, compiled programs *plus* language development tools – on two 92K diskettes. Pascal, my language of choice, simply wouldn't fit. Other languages, like Forth or C, did not appear to be available for the Osborne I at the time I began the project.

The use of an interactive language, like BASIC, made much of the develop-

ment easier than it might have been. There are, however, some big problems inherent in BASIC. Having only line numbers for labels and nothing but global variables made work very hard to follow in many places. Comments had to be kept to a minimum as they take up memory space.

In addition, Microsoft BASIC does not have multi-line, user-defined functions; it makes poor use of memory; and it has a poor symbol table layout. Each and every time a variable name is used in a program, BASIC stores the complete variable name (in ASCII), even though BASIC "tokenizes" much of the language. Consequently, variable names had to be kept short, which only compounded the problem of too few comments. On the bright side, BASIC-80 is fairly fast as far as interpreters go, and the BASIC-80 compiler does an excellent job of producing compiled code.

Compiler Availability

The compiled version of Augusta, which is 32K in length, compiles programs at well over 300 lines per minute (timed on an Osborne I – much faster compile speeds have been seen on systems using decent disk drives). Since there is no link step, compiled programs may be executed immediately.

The p-code interpreter requires just over 7K of memory and executes programs at a speed comparable to other p-code languages (see Table 2, page 34, *DDJ* No. 77, March 1983). In addition, Augusta p-code programs average about 30% fewer bytes of generated code than other p-code systems.

The Augusta compiler, including source, CP/M (Z80-based) p-code interpreter, a powerful debug utility, and a comprehensive 160-page reference guide is available from Laboratory Microsystems, Inc., 4147 Beethoven Street, Los Angeles, California 90066; (213) 306-7412.

Acknowledgements

Special thanks to Larry Backstrom who loaned me a video monitor for use with my Osborne I. Thanks also go to Jim Gilbert, and especially David Neal, who both provided a sounding board for several of my ideas, and provided many useful suggestions.

Loving appreciation is, of course, extended to my wife, Kim, for letting me work on this project during its seemingly endless development period. DDJ

(Tables I-IV on pages 20-22)

(Listing begins on page 22)

Table I.
Cross Reference of Major Subroutines

Line	Title And/Or Function
1000	Outer block of compiler (prompts for source and code files, and closes and updates the code file at exit)
1110	Initialization
1230	Open a new source file
1250	Convert CH$ to upper-case character
1280	GETLINE (read a source line from the input file)
1360	GETCH (sets CH$ to a character from the input)
1400	GETSYM (the lexical analyzer)
1780	Additional initialization, including reading the keywords data file
1890	Look up keyword in hash table KEY$()
1970	COMPILATION (from syntax diagrams)
2010	PARSE PROC (parses a procedure definition)
2100	PROCFORMALPART (parses procedure parameter list)
2160	PROCPARAMDECL (parses individual parameter declaration within the parameter list)
2250	SUBTYPEINDICATIONUNIT (assigns a "type" to a variable list)
2300	OBJSZ (determines the size in bytes of a variable or object – e.g., an array)
2340	PARSEFUNC (parses a function definition)
2440	BODYPART (see syntax diagrams in *DDJ* No. 75)
2480	DECLPART
2560	OBJDECL
2650	CONSTANT (constant declarations)
2700	ARRAY (array declarations)
2770	PRAGMA (initial parsing of pragmas)
2790	STMT (see STMT PART in syntax diagrams)
2810	SEQOFSTMTS (see SEQ OF STMTS in syntax diagrams)
2830	NULL statement
2850	BLOCK (optional DECLARE followed by BEGIN-END)
2890	EXIT statement
2930	RETURN statement
2970	IF statement
3100	EXPR (expression parsing)
3190	RELATION (relational subexpressions)
3290	SE (simple expression parsing including addition, subtraction)
3350	TERM (multiplication operators)
3420	SKIP ;
3440	ID (procedure calls or assignment statements)
3570	ACTUALPARAM (parameters being passed to a procedure or function)
3610	PRIMARY (expression parsing)
3635	Generate "load constant" instructions
3650	Generate " LCA" load string instruction
3710	Generate an array reference for integers, chars, booleans
3745	Parenthesized expression
3800	Generate a string array reference
3820	Generate "Load Value" instructions, including LDO, LDL, and LDU
3831	Global loads
3834	Local loads
3850	ADD ID (add identifier to symbol table)
3890	Look up ID$ in the symbol table
3990	Write a byte of code to the code file
4010	Read a word of code from the code file
4030	Write a word of code to the code file
4060	Generate load address instructions
4085	Global load addresses
4090	Local load addresses
4100	Generate call procedure opcodes
4140	GETBUF (locate a buffer to the code file)
4260	Read a byte from the code file
4280	Push X onto compiler's stack
4300	Pop top of compiler's stack into X
4320	Parse LOOP statements
4380	FOR loop
4580	WHILE loop
4590	Basic LOOP-END LOOP
4630	CASE statement
4810	OTHERS clause of CASE
4830	PRAGMA (additional processing for pragmas)
4910	WRITEPROC (creates an entry in the procedure table)
4930	Does type checking for booleans
4960	Does type checking for integers
4990	Keeps track of the maximum offset, or variable address
5020	Error handling

End Table I.

Table II.

Variable Dictionary

The dictionary includes those variables that are not used as "constant" symbols. All symbols are integers unless explicitly labelled as "real" using "!" or string using "$".

Variable Name	Use
ADDR	"Address" field of a symbol table entry
B()	Keeps track of which code file record matches which buffer
B$()	Holds the code file output buffers
B	Buffer pointer; points to next character in BUF$ to read
BUF$	Holds current source line from input file
CASES	Counts the number of cases in a case statement
CLST	Non-zero if listing on the CRT
CONST	The "constant" field in a symbol table entry
CPROC	Contains the number of the procedure being compiled
C$	Name of the code file
CH$	Holds the current input character from the source file
CB	Code Base points to first byte of code for a procedure; CP is an offset from CB
CP	Code Pointer; points to next byte of code in code file
D$	Dummy string variable
E	Set to the Error code value
GC	Global Code counter; counts the number of bytes of code
HASH	Used in looking up a keyword identifier
ID$	Holds an identifier name
KEY$()	An array holding all of the keyword identifiers
KIND	The "kind" field of a symbol table entry
LEX	The lexical level field of a symbol table entry
LL	The Lexical Level of the procedure being compiled
LOC1	Used in searching the symbol table
LOC2	Used in searching the symbol table
LFJP	Points to the head of a linked list of FJP instructions (used for backpatching)
LPFLG	Set to -1 if inside a LOOP-END LOOP, otherwise 0. EXIT is only allowed if LPFLG is non-zero
LUJP	Points to the head of a linked list of UJP instructions (used for backpatching)
MAP()	Hash "function" array
M0	Highest or maximally used record in the code file
MB	Maximum number of code file Buffers
MXOF	The highest variable offset used during the current procedure definition
OBJSZ	"Object Size" field of a symbol table entry
OFST	The current variable offset used during the current procedure definition
OLDB	Marks the start of a token; used in GETSYM
PINFO	"Parameter Info" field of a symbol table entry
PLST	Non-zero if listing to the printer
PROC	Counts the number of procedures in a program
R0	Set to the current code file buffer
R1	Code file record to write next byte of code
R2	Current offset within the code file buffer
S$()	Holds the symbol table as a stack of strings
S()	Internal stack space
SI	Source Input file number
SP	Internal Stack Pointer
S$	"string" returned by GETSYM
SSP	Symbol table Stack Pointer; points to top of S$() stack
T	Holds the Token "constant" returned by GETSYM
T0	Value of the token that is expected by the parser
TN	Token Number (holds the numeric equivalent of a constant read by GETSYM)
TSP	Internal "Type" Stack Pointer; used to parse expressions
TT$	Equal to CHR$(T)
TY()	Expression "type" checking stack
TYPE	"Type" field of a symbol table entry
V	Used to convert a word to two bytes
VLOC1	Address of byte 1 of variable V in memory; V is then accessed with PEEK()
VLOC2	Address of byte 2 of variable V
W	Holds the Word value to write out
X	The value pushed or popped onto the stack
X!	Real variable used in searching symbol table for integers > 32767
XITJP	Points to the head of a linked list of UJP instructions (used to EXIT a loop)

End Table II.

Table III.

Summary of Error Codes

Code	Meaning
1	Unrecognized character in source input
2	Undefined identifier
3	Syntax error in declaration part
4	Syntax error
5	CASE selector must be integer or character constant
6	Invalid pragma
7	Undefined attribute
8	Undefined type
9	Type mismatch
10	String constant exceeds source line
11	Invalid character constant
12	Unexpected end of input file
13	Semicolon missing – assumed present
14	EXIT is only allowed inside of LOOP-END LOOP

End Table III.

(Table IV begins on page 22)

Table IV.

"EXPECTED" Error Message codes, in the form,

nn EXPECTED

where nn is defined below.

Code	Token expected
0	' (single quote)
1	* not used *
2	integer constant
3	((left parenthesis)
4	) (right parenthesis)
5	*
6	/
7	+
8	-
9	<
10	<=
11	>
12	>=
13	=
14	/=
15	:
16	identifier
17	string constant
18	, (comma)
19	; (semicolon)
20	: (colon)
21	=>
22	:=
23	. (period)
24	.. (double period)
25	character constant as 'x'
26	@ symbol
27	AND
28	ARRAY
29	BEGIN
30	CASE
31	CONST
32	DECLARE
33	ELSE
34	ELSEIF
35	END
36	EXIT
37	FOR
38	FUNCTION
39	IF
40	IN
41	IS
42	LOOP
43	LAST
44	LENGTH
45	MOD
46	NOT
47	NULL
48	OF
49	OR
50	OTHERS
51	OUT
52	PRAGMA
53	PROCEDURE
54	RETURN
55	REVERSE
56	THEN
57	WHEN
58	WHILE

End Table IV.

Augusta Part IV

Listing One

(Text begins on page 12)

This is Part 2 of the Augusta Compiler listing, continued from the May, 1983 issue (*DDJ* No. 79, Augusta Part III, Listing Two, pages 20-26). This installment concludes the listing of the compiler.

Augusta Compiler, Copyright © 1983 by Computer Linguistics. Permission is granted to individuals to use the compiler for their personal use. Unauthorized distribution is expressly prohibited. Augusta is a trademark of Computer Linguistics.

```
3174 IF (TY(TSP)<>TBOL) OR
       (TY(TSP)<>TY(TSP-1)) THEN E=9:
       GOTO 5020
3175 TSP=TSP-1:
       GOSUB 4300:
       LFJP=X:
       GOSUB 4300:
       PREV=X:
       IF X=KAND+KTHEN THEN W=PAND ELSE W=POR
3178 GOSUB 3990:
       GOTO 3120
3180 T2=CP:
       WHILE LFJP<>0:
       CP=LFJP:
       GOSUB 4010:
       LFJP=W:
       W=T2-CP-2:
       GOSUB 4030:
       WEND:
       CP=T2:
       RETURN


3190 'RELATION
3200 GOSUB 3300
3210 IF INSTR(RELOPS$,TT$)=0 THEN RETURN
3220 X=T:
       GOSUB 4280:
       GOSUB 1400
3230 GOSUB 3290:
       IF TY(TSP)<>TINT AND TY(TSP)<>TCHR AND
       TY(TSP)<>TBOL GOTO 3260
3235 IF TY(TSP)<>TY(TSP-1) THEN E=9:
       GOTO 5020 ELSE TSP=TSP-1:
       TY(TSP)=TBOL
3240 GOSUB 4300:
       IF X=LES THEN W=PLESI ELSE IF X=LEQ
       THEN W=PLEQI ELSE IF X=GT THEN W=PGTRI
3245 IF X=GEQ THEN W=PGEQI ELSE IF X=EQ
       THEN W=PEQUI ELSE IF X=NEQ THEN W=PNEQI
3250 GOSUB 3990:
       GOTO 3210
3260 IF TY(TSP)<>TSTR OR
       TY(TSP)<>TY(TSP-1) THEN E=9:
       GOTO 5020 ELSE TSP=TSP-1:
       TY(TSP)=TBOL
3270 GOSUB 4300:
       IF X=LES THEN W=PLESSTR ELSE IF X=LEQ
       THEN W=PLEQSTR ELSE IF X=GT THEN
       W=PGTRSTR
3275 IF X=GEQ THEN W=PGEQSTR ELSE IF
       X=EQ THEN W=PEQUSTR ELSE IF X=NEQ THEN
       W=PNEQSTR
3280 GOSUB 3990:
       GOTO 3210


3290 'SE
3300 IF INSTR(UNARYOPS$,TT$) THEN X=T:
       GOSUB 4280:
       X=1:
       GOSUB 4280:
       GOSUB 1400 ELSE X=0:
       GOSUB 4280
3310 GOSUB 3350:
       GOSUB 4300:
       IF X=1 THEN GOSUB 4300:
       IF X=SUBT THEN W=PNGI:
       GOSUB 3990 ELSE W=PNOT:
       GOSUB 3990
3320 IF INSTR(ADDOPS$,TT$)=0 THEN RETURN
3330 X=T:
       GOSUB 4280:
       GOSUB 1400:
       GOSUB 3350:
       GOSUB 4300:
       IF X=ADD THEN W=PADI ELSE W=PSBI
3340 IF TY(TSP)<>TINT THEN E=9:
       GOTO 5020 ELSE TSP=TSP-1:
       GOSUB 3990:
       GOTO 3320



3350 'TERM
3360 GOSUB 3610
3370 IF INSTR(MULOPS$,TT$)=0 THEN RETURN
3380 X=T:
       GOSUB 4280:
       GOSUB 1400:
       GOSUB 3610
3390 IF (TY(TSP)<>TY(TSP-1)) OR
       (TY(TSP)<>TINT) THEN E=9:
       GOTO 5020 ELSE TSP=TSP-1
3400 GOSUB 4300:
       IF X=MUL THEN W=PMPI ELSE IF X=DIV THEN
       W=PDVI ELSE W=PMODI
```

(Continued on page 24)

Augusta Part IV

Listing One

(Listing continued, text begins on page 12)

```
3410 GOSUB 3990:
       GOTO 3370



3420 'SKIP ;
3430 IF T=SEMICOLON THEN GOSUB 1400:
       RETURN ELSE E=13:
       GOSUB 5110:
       RETURN



3440 'ID
3450 GOSUB 3890:
       IF KIND<>2 THEN X=TYPE:
       GOSUB 4280:
       GOSUB 3490:
       GOTO 3530 ELSE X=ADDR:
       GOSUB 4280:
       X=LEX:
       GOSUB 4280


3460 GOSUB 1400:
       IF T=SEMICOLON GOTO 3480 ELSE T0=LP:
       GOSUB 1960
3470 GOSUB 3570:
       T0=RP:
       GOSUB 1960
3480 GOSUB 4100:
       GOSUB 3420:
       RETURN
3490 GOSUB 4060:
       GOSUB 1400
3500 IF KIND<>1 THEN RETURN ELSE X=OBJSZ:
       GOSUB 4280
3510 T0=LP:
       GOSUB 1960:
       GOSUB 3100:
       GOSUB 4960:
       GOSUB 4300:
       IF X=2 THEN W=PIND ELSE W=PIXA:
       GOSUB 3990:
       W=X
3520 GOSUB 3990:
       T0=RP:
       GOSUB 1960:
       RETURN
3530 T0=COLONEQ:
       GOSUB 1960
3540 GOSUB 3100:
       GOSUB 4300:
       IF NOT(X=TY(TSP) OR (X=TINT AND
       TY(TSP)=TBOL) OR (X=TBOL AND
       TY(TSP)=TINT)) THEN E=9:
       GOTO 5020
3550 IF X=TSTR THEN W=PSAS ELSE W=PSTO
3560 TSP=TSP-1:
       GOSUB 3990:
       GOSUB 3420:
       RETURN
```

```
3570 'ACTUALPARAM
3580 IF T=AT THEN GOSUB 1400:
       T0=ID:
       GOSUB 1950:
       GOSUB 3890:
       GOSUB 3490 ELSE GOSUB 3100:
       TSP=TSP-1
3590 IF T=COMMA THEN GOSUB 1400:
       GOTO 3580
3600 RETURN



3610 'PRIMARY
3620 IF T=LP THEN GOSUB 1400:
       GOSUB 3100:
       T0=RP:
       GOSUB 1960:
       RETURN



3630 IF T=C THEN TSP=TSP+1:
       TY(TSP)=TINT:
       GOSUB 3640:
       GOSUB 1400:
       RETURN
3633 IF T=CH THEN TSP=TSP+1:
       TY(TSP)=TCHR:
       GOSUB 3640:
       GOSUB 1400:
       RETURN ELSE GOTO 3650



3635 'LD CONS
3640 IF TN=-1 THEN W=PSLDCN1:
       GOTO 3645 ELSE IF TN>-1 AND TN<16 THEN
       W=64+TN:
       GOTO 3645
3643 IF TN>0 AND TN<256 THEN W=PSLDC:
       GOSUB 3990:
       W=TN:
       GOSUB 3990:
       RETURN ELSE W=PLDCI:
       GOSUB 3990:
       W=TN:
       GOSUB 4030:
       RETURN
3645 GOSUB 3990:
       RETURN
3650 IF T<>SC GOTO 3670 ELSE TSP=TSP+1:
       TY(TSP)=TSTR
3660 W=PLCA:
       GOSUB 3990:
       W=LEN(S$):
       GOSUB 3990:
       FOR I=1 TO LEN(S$):
       W=ASC(MID$(S$,I)):
       GOSUB 3990:
       NEXT I:
       GOSUB 1400:
       RETURN
3670 T0=ID:
       GOSUB 1950
3680 GOSUB 3890:
       IF KIND=0 THEN TSP=TSP+1:
       TY(TSP)=TYPE:
       TN=CONST:
       GOSUB 3640:
       GOSUB 1400:
       RETURN
```

```
3682 GOSUB 1400:
       IF T=SQUOTE GOTO 3780
3685 IF KIND=4 THEN X=TYPE:
       GOSUB 4280:
       T0=LP:
       GOSUB 1960:
       GOSUB 3100:
       T0=RP:
       GOSUB 1960:
       GOSUB 4300:
       TY(TSP)=X:
       RETURN



3690 TSP=TSP+1:
       TY(TSP)=TYPE:
       IF TYPE=0 GOTO 3800
3700 IF KIND<>1 GOTO 3740 ELSE GOSUB 4060
3710 T0=LP:
       GOSUB 1960
3720 GOSUB 3100:
       IF TY(TSP)<>TINT THEN E=9:
       GOTO 5020 ELSE TSP=TSP-1:
       W=PIND:
       GOSUB 3990:
       W=PSINDO:
       GOSUB 3990
3730 T0=RP:
       GOSUB 1960:
       RETURN
3740 IF KIND<>3 THEN GOSUB 3760:
       RETURN ELSE X=ADDR:
       GOSUB 4280:
       X=LEX:
       GOSUB 4280
3745 IF T=LP THEN GOSUB 1400:
       GOSUB 3570:
       T0=RP:
       GOSUB 1960
3750 GOSUB 4100:
       RETURN
3760 GOSUB 3820:
       IF PINFO=2 THEN W=PSINDO:
       GOSUB 3990
3770 RETURN
3780 TSP=TSP+1:
       TY(TSP)=TINT:
       GOSUB 1400:
       IF T=KLAST THEN W=PLDCI:
       GOSUB 3990:
       W=CONST:
       GOSUB 4030:
       GOTO 3790
3785 IF T=KLEN THEN GOSUB 4060:
       W=PLDB:
       GOSUB 3990 ELSE E=7:
       GOTO 5020
3790 GOSUB 1400:
       RETURN
3800 IF KIND<>1 GOTO 3810 ELSE GOSUB 4060:
       X=OBJSZ:
       GOSUB 4280:
       T0=LP:
       GOSUB 1960:
       GOSUB 3100
3805 IF TY(TSP)<>TINT THEN E=9:
       GOTO 5020 ELSE TSP=TSP-1:
       GOSUB 4300:
       W=PIXA:
       GOSUB 3990:
       W=X:
       GOSUB 3990:
```

```
      T0=RP:
      GOSUB 1960:
      RETURN
3810 GOSUB 4060:
      RETURN


3820 'LD VAL
3830 IF LEX=1 GOTO 3831 ELSE IF LEX=LL
      GOTO 3834 ELSE W=PLOD:
      GOSUB 3990:
      W=LL-LEX:
      GOTO 3845


3831 'GLOBAL
3832 IF ADDR<256 THEN W=PSLDO:
      GOTO 3840 ELSE W=PLDO:
      GOTO 3845


3834 'LDL
3835 IF ADDR>=0 AND ADDR<8 THEN W=PSLDL0+ADDR:
      GOSUB 3990:
      RETURN
3836 IF ADDR>0 AND ADDR<8 THEN W=PSLDL:
      GOTO 3840 ELSE W=PLDL:
      GOTO 3845



3840 'B,B
3842 GOSUB 3990:
      W=ADDR:
      GOSUB 3990:
      RETURN



3845 'B,W
3847 GOSUB 3990:
      W=ADDR:
      GOSUB 4030:
      RETURN



3850 'ADD ID
3860 IF LEN(S$(SSP))+17>255 THEN SSP=SSP+1
3870 S$(SSP)=ID$+CHR$(TYPE)+CHR$(KIND)+
      CHR$(PINFO)+MKI$(CONST)+CHR$(OBJSZ)+
      MKI$(ADDR)+CHR$(LL)+S$(SSP)
3880 RETURN



3890 'LOOKUP ID
3900 LOC1=SSP
3910 LOC2=INSTR(S$(LOC1),ID$):
      IF LOC2 GOTO 3920 ELSE LOC1=LOC1-1:
      IF LOC1 GOTO 3910 ELSE E=2:
      GOTO 5020
3920 T9=VARPTR(S$(LOC1)):
      POKE VLOC,PEEK(T9+1):
      POKE VLOC1,PEEK(T9+2):
      T9=V+LOC2-1
3930 TYPE=PEEK(T9+8):
      KIND=PEEK(T9+9):
      PINFO=PEEK(T9+10):
      POKE VLOC,PEEK(T9+11):
      POKE VLOC1,PEEK(T9+12):
      CONST=V
3960 OBJSZ=PEEK(T9+13):
      LEX=PEEK(T9+16):
      POKE VLOC,PEEK(T9+14):
      POKE VLOC1,PEEK(T9+15):
      ADDR=V:
      RETURN



3990 'GENBYTE
4000 GOSUB 4140:
      FIELD #1,R2 AS D$,1 AS D$:
      LSET D$=CHR$(W):
      CP=CP+1:
      RETURN



4010 'READWRD
4020 T1=CP:
      GOSUB 4260:
      POKE VLOC,W:
      CP=CP+1:
      GOSUB 4260:
      POKE VLOC1,W:
      W=V:
      CP=T1:
      RETURN



4030 'GENWORD W
4040 GOSUB 4140:
      IF R2<127 THEN FIELD #1, R2 AS D$,2 AS D$:
      LSET D$=MKI$(W):
      CP=CP+2:
      RETURN
4050 V=W:
      W=PEEK(VLOC):
      GOSUB 3990:
      W=PEEK(VLOC1):
      GOSUB 3990:
      RETURN
4060 'LD ADR
4070 IF PINFO=2 THEN GOSUB 3820:
      RETURN
4080 IF LEX=1 GOTO 4085 ELSE IF LEX=LL
      GOTO 4090 ELSE W=PLDA:
      GOSUB 3990:
      W=LL-LEX:
      GOTO 3845



4085 'GL ADR
4087 IF ADDR<256 THEN W=PSLAO:
      GOTO 3840 ELSE W=PLAO:
      GOTO 3845



4090 'LDL ADR
4095 IF ADDR>=0 AND ADDR<256 THEN W=PSLLA:
      GOTO 3840 ELSE W=PLLA:
      GOTO 3845



4100 'CALL PROC
4110 GOSUB 4300:
      LEX=X:
      GOSUB 4300:
      ADDR=X
4120 IF LEX=0 THEN W=PCSP ELSE IF LEX=2
      THEN W=PCSP ELSE IF LEX=LL+1 THEN
      W=PCLP ELSE W=PCIP
4130 GOSUB 3990:
      W=ADDR:
      GOSUB 3990:
      RETURN



4140 'GETBUF
4150 T9=CP+CB:
      R1=T9\128+1:
      R2=T9 AND 127:
      IF R1=R0 THEN RETURN
4160 FIELD #1,128 AS D$:
      J=1
4170 IF B(J)=R0 OR B(J)=0 GOTO 4190 ELSE J=J+1:
      IF J<=MB GOTO 4170
4180 LSET B$(0)=D$:
      J=INT(RND*MB)+1:
      LSET D$=B$(J):
      PUT #1,B(J):
      LSET B$(J)=B$(0):
      B(J)=R0:
      GOTO 4200
4190 LSET B$(J)=D$:
      B(J)=R0
4200 J=1
4210 IF B(J)=R1 GOTO 4240 ELSE J=J+1:
      IF J<=MB GOTO 4210
4220 GET #1, R1:
      R0=R1:
      IF R1>M0 THEN M0=R1
4230 RETURN
4240 LSET D$=B$(J):
      R0=R1:
      IF R1>M0 THEN M0=R1
4250 RETURN



4260 'READBYTE
4270 GOSUB 4140:
      FIELD#1,R2 AS D$,1 AS D$:
      W=ASC(D$):
      RETURN



4280 'PUSH
4290 S(SP)=X:
      SP=SP+1:
      RETURN



4300 'POP
4310 SP=SP-1:
      X=S(SP):
      RETURN



4320 'LOOP
4330 IF T<>KWHILE GOTO 4370
4340 GOSUB 1400:
      X=CP:
      GOSUB 4280:
      GOSUB 3100:
      GOSUB 4930
4350 W=PFJP:
```

(Continued on page 28)

Augusta Part IV

Listing One

(Listing continued, text begins on page 12)

```
      GOSUB 3990:
      X=CP:
      GOSUB 4280:
      W=0:
      GOSUB 4030:
      GOSUB 4590:
      GOSUB 4300
4360 T1=CP:
      CP=X:
      W=T1-X+1:
      GOSUB 4030:
      CP=T1:
      W=PUJP:
      GOSUB 3990:
      GOSUB 4300:
      W=X-CP-2:
      GOSUB 4030:
      GOSUB 4620:
      RETURN
4370 IF T<>KFOR GOTO 4580
4380 GOSUB 1400:
      T0=ID:
      GOSUB 1950:
      X=OFST:
      GOSUB 4280:
      GOSUB 5400
4390 ADDR=OFST:
      TYPE=1:
      KIND=5:
      PINFO=0:
      GOSUB 3850
4400 GOSUB 1400:
      T0=KIN:
      GOSUB 1960
4410 IF T=KREVERSE THEN X=-1:
      GOSUB 1400 ELSE X=1
4420 GOSUB 4280:
      W=PLLA:
      GOSUB 3990:
      W=OFST:
      GOSUB 4030
4430 GOSUB 3290:
      GOSUB 4960:
      W=PSTO:
      GOSUB 3990
4440 X=CP:
      GOSUB 4280:
      W=PLDL:
      GOSUB 3990:
      W=OFST:
      GOSUB 4030
4450 T0=DOTDOT:
      GOSUB 1960:
      GOSUB 3290:
      GOSUB 4960

4460 GOSUB 4300:
      T1=X:
      GOSUB 4300:
      IF X<0 THEN W=PGEQI ELSE W=PLEQI
4470 GOSUB 3990:
      W=PFJP:
      GOSUB 3990:
      GOSUB 4280:
      X=T1:
      GOSUB 4280
4480 X=CP:
      GOSUB 4280:
      W=0:
      GOSUB 4030:
      X=OFST:
      GOSUB 4280:
      OFST=OFST+2:
      GOSUB 4990

4490 GOSUB 4590:
      GOSUB 4300:
      T3=X:
      GOSUB 4300:
      T1=X:
      GOSUB 4300:
      T2=X:
      GOSUB 4300:
      IF X<0 THEN W=PDECL ELSE W=PINCL
4500 GOSUB 3990:
      W=T3:
      GOSUB 4030
4520 W=PUJP:
      GOSUB 3990
4530 W=T2-CP-2:
      GOSUB 4030:
      T2=CP:
      CP=T1:
      W=T2-T1-2:
      GOSUB 4030:
      CP=T2
4550 GOSUB 5500:
      GOSUB 5700
4560 GOSUB 4300:
      OFST=X
4570 GOSUB 4620:
      RETURN
4580 X=CP:
      GOSUB 4280:
      GOSUB 4590:
      W=PUJP:
      GOSUB 3990:
      GOSUB 4300:
      W=X-CP-2:
      GOSUB 4030:
      GOSUB 4620:
      RETURN
4590 T0=KLOOP:
      GOSUB 1960:
      X=XITJP:
      GOSUB 4280:
      XITJP=0:
      X=LPFLG:
      GOSUB 4280:
      LPFLG=-1:
      GOSUB 2810
4600 T0=KEND:
      GOSUB 1960
4610 T0=KLOOP:
      GOSUB 1960:
      GOSUB 4300:
      T5=X:
      GOSUB 4300:
      T6=X:
      GOSUB 3420:
      RETURN
4620 T2=CP:
      WHILE XITJP<>0:
      CP=XITJP:
      GOSUB 4010:
      XITJP=W:
      W=T2-CP-2:
      GOSUB 4030:
      WEND:
      CP=T2:
      LPFLG=T5:
      XITJP=T6:
      RETURN


4630 'CASE
4640 GOSUB 1400:
      GOSUB 3100:
      IF TY(TSP)<>TINT AND TY(TSP)<>TCHR THEN
      E=9:
      GOTO 5020
4645 TSP=TSP-1:
      W=PXJP:
      GOSUB 3990:
      X=CP:
      GOSUB 4280:
      GOSUB 4030:
      GOSUB 4030:
      GOSUB 4030:
      CASES=0:
      LUJP=0:
      T0=KIS:
      GOSUB 1960
4650 T0=KWHEN:
      GOSUB 1960:
      IF T=KOTHERS GOTO 4810 ELSE T1=0
4660 IF T=ID THEN GOSUB 3890:
      TN=CONST:
      IF TYPE=1 OR TYPE=2 THEN T=C
4670 IF T<>CH AND T<>C THEN E=5:
      GOTO 5020 ELSE X=TN:
      GOSUB 4280:
      T1=T1+1:
      GOSUB 1400:
      IF T=BAR THEN GOSUB 1400:
      GOTO 4660
4680 GOSUB 4780
4690 IF T=KWHEN GOTO 4650 ELSE X=0:
      GOSUB 4280:
      GOSUB 4280:
      X=1:
      GOSUB 4280:
      CASES=CASES+1
4700 T0=KEND:
      GOSUB 1960:
      T0=KCASE:
      GOSUB 1960
4710 T1=SP-4:
      T3=32767:
      T4=-32767:
      FOR I=1 TO CASES-1:
      T2=S(T1):
      T1=T1-2:
      FOR J=1 TO T2:
      IF S(T1)<T3 THEN T3=S(T1)
4715 IF S(T1)>T4 THEN T4=S(T1)
4720 T1=T1-1:
      NEXT J:
      NEXT I:
      W=PUJP:
      GOSUB 3990:
```

```
      T5=CP:
      GOSUB 4300:
      GOSUB 4300:
      T1=X:
      GOSUB 4300

4725 IF X=-1 THEN W=T1-CP-2:
      GOSUB 4030 ELSE W=LUJP:
      LUJP=CP:
      GOSUB 4030
4730 FOR I=T3 TO T4:
      W=T5-CP-3:
      GOSUB 4030:
      NEXT I'BUILD TABLE
4740 T7=CP:
      FOR I=1 TO CASES-1:
      GOSUB 4300:
      T2=X:
      GOSUB 4300:
      T6=X:
      FOR T8=1 TO T2:
      GOSUB 4300
4745 CP=T5+(X-T3)*2+2:
      W=T6-CP-2:
      GOSUB 4030:
      NEXT T8:
      NEXT I:
      CP=T7
4750 GOSUB 4300:
      T2=CP:
      CP=X:
      W=T3:
      GOSUB 4030:
      W=T4:
      GOSUB 4030:
      W=T5-CP-2:
      GOSUB 4030
4760 WHILE LUJP<>0:
      CP=LUJP:
      GOSUB 4010:
      LUJP=W:
      W=T2-CP-2:
      GOSUB 4030:
      WEND:
      CP=T2
4770 GOSUB 3420:
      RETURN
4780 T0=EQGT:
      GOSUB 1960:
      X=CP:
      GOSUB 4280:
      X=T1:
      GOSUB 4280:
      X=LUJP:
      GOSUB 4280:
      CASES=CASES+1:
      X=CASES:
      GOSUB 4280:
      GOSUB 2810
4790 W=PUJP:
      GOSUB 3990:
      GOSUB 4300:
      CASES=X:
      GOSUB 4300:
      LUJP=X
4800 W=LUJP:
      LUJP=CP:
      GOSUB 4030:
      RETURN

4810 'OTHERS
4820 GOSUB 1400:
      X=-1:
      GOSUB 4280:
      T1=1:
      GOSUB 4780:
      GOTO 4700

4830 'PRAGMA
4840 GOSUB 1400:
      IF S$()"LIST" GOTO 4850
4845 GOSUB 4880:
      IF T$="ON" THEN PLST=-1:
      LPRINT LP$;:
      RETURN ELSE IF T$="OFF" THEN PLST=0:
      RETURN ELSE E=6:
      GOTO 5020
4850 IF S$="CRT" THEN GOSUB 4880:
      IF T$="ON" THEN CLST=-1:
      RETURN ELSE CLST=0:
      RETURN
4860 IF S$<>"INCLUDE" THEN RETURN ELSE GOSUB 1400:
      T0=LP:
      GOSUB 1960
4870 IF T<>SC THEN E=9:
      GOTO 5020 ELSE GOSUB 1230:
      GOSUB 1400:
      T0=RP:
      GOSUB 1960:
      RETURN
4880 GOSUB 1400:
      T0=LP:
      GOSUB 1960:
      T$=S$:
      GOSUB 1400:
      T0=RP:
      GOSUB 1960:
      RETURN

4910 'WRITEPROC
4920 T2=CP:
      T3=CB:

      CB=0:
      CP=(ADDR-1)*7+128:
      W=C1-1920:
      GOSUB 4030:
      W=L1:
      GOSUB 4030:
      W=P1:
      GOSUB 4030:
      W=LL:
      GOSUB 3990:
      CP=T2:
      CB=T3:
      RETURN

4930 'CHEK BOOL
4940 IF TY(TSP)<>TBOL THEN E=9:
      GOTO 5020
4950 TSP=TSP-1:
      RETURN

4960 'CHEK INT
4970 IF TY(TSP)<>TINT THEN E=9:
      GOTO 5020
4980 TSP=TSP-1:
      RETURN

4990 'MAX OFFST
5000 IF OFST>MXOF THEN MXOF=OFST
5010 RETURN
     GOSUB 5100:
      STOP
5100 PRINT:
      PRINT"ERROR";E;"AT LINE";LN:
      PRINT BUF$:
      PRINT TAB(B-1);"*":
      RETURN
5110 PRINT:
      PRINT T0;" EXPECTED":
      GOSUB 5100:
      RETURN

5200 'PROC DEF
5210 LL=LL+1:
      X=CPROC:
      GOSUB 4280:
      X=OFST:
      GOSUB 4280:
      X=MXOF:
      GOSUB 4280:
      T0=ID:
      GOSUB 1950
5220 GOSUB 5400:
      RETURN

5300 'PROC END DEF
5310 W=PEOP:
      GOSUB 3990:
      GOSUB 4300:
      P1=X:
      GOSUB 4300:
      ADDR=X:
      CPROC=X:
      L1=MXOF:
      C1=GC:
      GOSUB 4910:
      GC=GC+CP
5320 LL=LL-1:
      GOSUB 5500:
      GOSUB 5600
5330 GOSUB 4300:
      MXOF=X:
      GOSUB 4300:
      OFST=X:
      GOSUB 4300:
      CPROC=X:
      RETURN
```

(Continued on page 32)

Augusta Part IV

Listing One

(Listing continued, text begins on page 12)

```
5400 'PUSH SYMS
5410 X=LEN(S$(SSP)):
       IF X=255 THEN SSP=SSP+1:
       X=0
5420 GOSUB 4280:
       X=SSP:
       GOSUB 4280:
       RETURN



5500 'POPSYMS
5510 GOSUB 4300:
       FOR I=X+1 TO SSP:
       S$(I)="":
       NEXT I:
       SSP=X:
       GOSUB 4300:
        LOC2=X:
        RETURN
 5520 RETURN
 5600 S$(SSP)=RIGHT$(S$(SSP),LOC2+17):
        RETURN
 5700 S$(SSP)=RIGHT$(S$(SSP),LOC2):
        RETURN
 32767 END
```

End Listing One

Listing Two

The "KEYWORDS.TXT" data file is read by the compiler to initialize its table of keywords and built-in procedures and functions. The numbers near the beginning of the file are read into the array MAP() and used as a hash function to access the keywords in array KEY$().

```
Augusta Keywords,(C)Cop··· Computer
15                  Linguistics 1983
0
0123456789
0123456789ABCDEF
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwyxz
ABCDEFGHIJKLMNOPQRSTUVWYXZ0123456789_
1
3
4
6
7
11
0
0
13
0
0
16
19
20
22
26
0
28
0
30
0
0
31
0
0
0
FILEARRA,5,4,0,,50,,
STRARRAY,0,4,0,,81,,
CHARRAY,2,4,0,,2,,
INTARRAY,1,4,0,,2,,
BOOLARRA,4,4,0,,2,,
TRUE,4,0,,-1,,,
FALSE,4,0,,0,,,
MININT,1,0,,-32767,,,
MAXINT,1,0,,32767,,,
INOUTFIL,5,4,0,,50,,
KEYPRESS,4,3,,,,40,0
SEARCH,1,3,,,,39,0
STR,,2,,,,38,0
VALUE,1,3,,,,37,0
INITFILE,,2,,,,36,0
ASSIGN,,2,,,,35,0
APPEND,,2,,,,34,0
FREEMEM,,2,,,,33,0
GETMEM,1,3,,,,32,0
IOERROR,1,3,,,,31,0
PUTBOOL,,2,,,,30,0
SKIPLINE,,2,,,,29,0
CHAR,2,3,,,,28,0
NEXTWRIT,1,3,,,,27,0
SETWRITE,,2,,,,26,0
NEXTREAD,1,3,,,,25,0
SETREAD,,2,,,,24,0
WRITE,,2,,,,23,0
READ,,2,,,,22,0
ISOPEN,4,3,,,,21,0
RENAME,1,3,,,,20,0
ERASE,1,3,,,,19,0
CLOSE,,2,,,,18,0
CREATE,,2,,,,17,0
OPEN,,2,,,,16,0
SCANB,1,3,,,,15.0
SCANF,1,3,,,,14,0
MOVERIGH,,2,,,,13,0
MOVELEFT,,2,,,,12,0
SUBSTR,,2,,,,11,0
POKE,,2,,,,10,0
PEEK,1,3,,,,9,0
PUTSTR,,2,,,.8,0
NEWLINE,,2,,,,7,0
PUTCHAR,,2,,,.6,0
GETCHAR,,2,,,,5,0
PUTINT,,2,,,,4,0
GETINT,,2,,,,3,0
PUTLINE,,2,,,,2,0
GETSTR,,2,,,,1,0
STRING,0,4,1,,81,,
INTEGER,1,4,1,,2,,
BOOLEAN,4,4,1,,2,,
CHARACTE,2,4,1,,2,,
$END$,,,,,,,
AND
ARRAY
BEGIN
CASE
CONSTANT
DECLARE
ELSE
ELSEIF
END
EXIT
FOR
FUNCTION
IF
IN
IS
LOOP
LAST
LENGTH
MOD
NOT
NULL
OF
OR
OTHERS
OUT
PRAGMA
PROCEDURE
RETURN
REVERSE
THEN
WHEN
WHILE
```

End Listing Two

RED
A Better C Screen Editor, Part I

by Edward K. Ream

RED is a descendant of the public-domain text editor (called ED2) which I described in the January 1982 issue of *Dr. Dobb's Journal*. I started work on RED so that I could edit files without worrying how large they were. RED's most important new feature is a set of new buffer routines which remove all restrictions on the size of files which may be edited.

RED appears just the same to the user as does ED2. The differences between RED and ED2 are all under the covers, and it is on these programming details that this article will dwell. Because of the length of RED's listing, this article will be split into two parts. This month I'll discuss how RED's buffer routines work. These routines use a programming technique called *virtual memory,* which I'll explain in detail. I'll also discuss a new method for recovering from errors which saves a lot of coding. Part I of this article concludes with a discussion of the performance of RED's new buffer routines.

The new buffer routines require unbuffered, i.e., sector-at-a-time, I/O routines. Rather than write my own, I decided to modify the assembly language portion of the BDS C run-time library. Leor Zolman, the author of BDS C, has generously given us all permission to use these routines in any way we choose. This new library will be presented in greater detail next month in Part II of this article.

New Buffer Routines

RED'S new buffer routines really got started when I bought (for 8 dollars!) a copy of the "Just Like Mom's Editor" from the BDS C Users' Group. The editor is still available and is in the public domain. I would like to thank Scott Fluhrer and Neal Somos for their work on this editor.

RED's buffer routines are direct descendants of the buffer routines that Fluhrer and Somos wrote. After a few hours work, I converted their routines to work with the BDS C version of RED. I quickly discovered two problems with the code. First, though files larger than memory could be handled, there was still a limit on the size of the files. Second, the buffer routines did not keep enough information about the location of the current line. As a result, scrolling the screen was terribly slow.

This was the start of what has become four new versions of the buffer routines. I used the excellent BDS C compiler for the first three versions. For the fourth version, I converted the buffer routines from BDS C to Small C. Most of the conversion work consisted of changing structures to parallel arrays.

Now let's see how the new buffer routines work. The original file being edited, the *user file,* must be copied to the buffer before editing can be done. When editing is complete, the save command or resave command is used to copy the buffer back to the user file.

ED2's buffer routines kept the buffer entirely in memory. RED's buffer routines keep most of the buffer on a disk file, the *work file,* which is used only by the editor; the work file is created when the buffer is initialized and it is erased when the editor ceases execution.

The work file is made up of a sequence of fixed-length *blocks;* it is not just a sequence of characters. A block is a multiple of the sector size of the disk. (The READ_SIZE constant gives the number of sectors per block.) The new buffer routines use the following three operations on the blocks and sectors of the work file:

sysseek (fd, block)
int fd, block;

Positions the file whose descriptor is fd to the indicated block. The next read or write operation will be to the block. Sysseek() can also be used to extend the file, i.e., to increase the size of the file. If the block number is exactly one larger than the number of the last block of the file, the file is made larger. Some operating systems do not allow files to be made larger dynamically. They require that the size of the file be known in advance. However, many operating systems (including Unix and CP/M) do allow files to be extended in this way.

sysread (fd, buffer)
int fd;
char * buffer;

Reads the current block into the block-sized buffer.

syswrite (fd, buffer, n):
int fd;
char * buffer;
int n;

Writes n sectors from the buffer to the file. When writing to the work file, n is always READ_SIZE. Thus, only full blocks are ever transferred to and from the work file.

The operating system does not have to provide exactly these primitives, but the operating system must make it possible to write them. This is the reason for the new run-time library in the Small C version of RED. Next month I will discuss these routines in more detail.

Inserting and Deleting Blocks

The CP/M operating system, like most operating systems, does not allow blocks or sectors to be inserted into the middle of a file or deleted from the middle of the file. CP/M does allow files to be extended after the file has been opened, but only at the end of the file. Thus, when sysseek() extends the file, the newly created sectors must follow immediately after the last sector which has been allocated.

Fortunately, these restrictions do not really prevent the editor from inserting and deleting blocks where required. For most purposes, the buffer routines can ignore the actual location of each block. The buffer routines link together the blocks in a list. As far as the buffer routines are concerned, the first block in the list *is* the first block of the work file, regardless of its actual location.

Each block contains two pointer fields: d_next, which points to the next block on the list, and d_back, which points to the previous block. Each pointer is an actual disk address; it is used directly by the sysseek primitive. A negative pointer indicates the end of the list. Two global variables, b_head and b_tail, point to the first and last blocks of the list.

Figure 1 (page 35) shows a typical list of blocks. Links are shown as arrows, except that negative links are shown as a slash. The values in the link fields are not

Edward K. Ream, 1850 Summit Avenue, Madison, Wisconsin 53705.

Copyright © 1983 by Edward K. Ream. RED may be used for non-commercial purposes only. No commercial use of RED may be made without the author's expressed written permission.

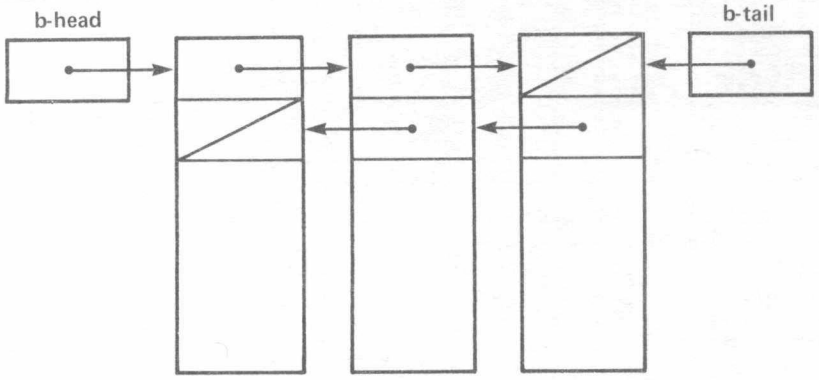
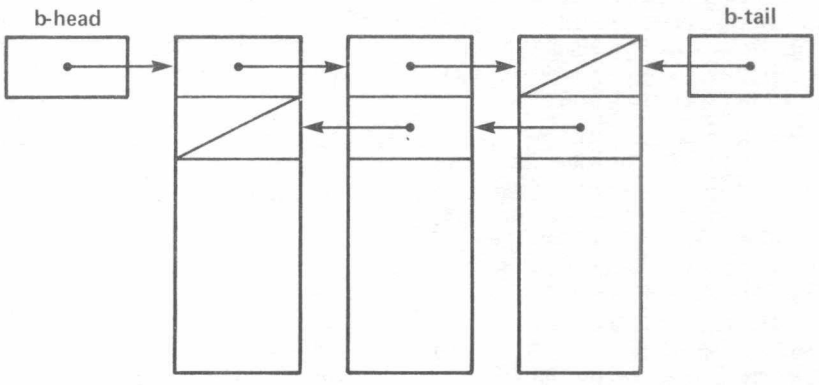


Figure 1.

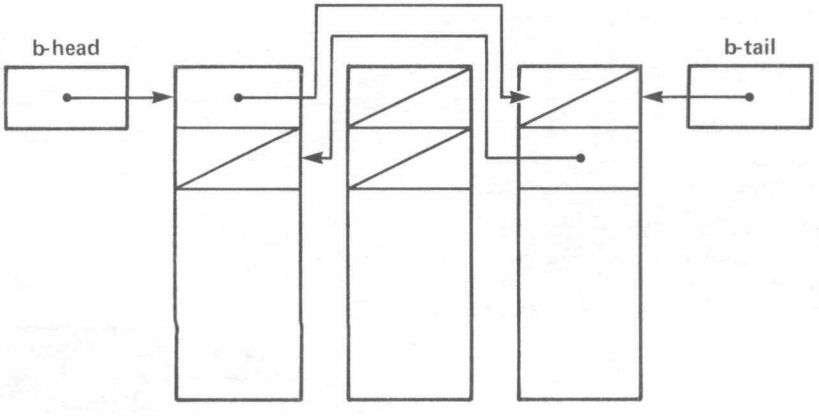
The blocks of the work file form a double-linked list.



Deleting blocks: before
Inserting blocks: after

Figure 2a.

Inserting a block in the work file is just the reverse of deleting a block.



Deleting blocks: after
Inserting blocks: before

Figure 2b.

shown because the actual location of each block on the disk does not matter. The blocks of the work file are kept on a doubly linked list. This makes inserting and deleting blocks much easier. Searching the list backwards is also made possible.

Figures 2a and 2b (at left) show how to insert or delete blocks from the work file. To delete a block, the pointers to the preceding and following blocks are changed so that nothing points to the deleted block. Figure 2a shows the work file before the block is deleted and Figure 2b shows the work file afterwards. Inserting a block is essentially the reverse of deleting a block. Considering Figures 2a and 2b in reverse order shows how a block is inserted into the work file.

Deleting the last block of the work file is a special case, illustrated in Figures 3a and 3b (page 36). However, it turns out that the first block of the list never changes. It is never deleted, nor is a block ever inserted before it. Thus, the b_head variable never changes; it could be replaced with the constant zero.

By now it should be obvious that nothing really gets deleted from the work file. Only the pointers to "deleted" blocks are erased. In order to keep track of all deleted blocks, they are kept on a singly linked *free list*. The global variable b_free points to the first block of the free list. Note that the blocks on the free list are still part of the work file as far as the operating system is concerned.

All changes to the free list are made at its head, as shown in Figures 4a and 4b (page 38). Figure 4a shows the list before a block is deleted while Figure 4b shows the block afterwards. Inserting a block into the free list is essentially the reverse process.

When inserting a block into the work file, a new block is taken from the free list if the free list is not empty. Otherwise, the sysseek() routine is used to allocate a new block from the disk. The variable b_max_diskp keeps track of the last block that has been allocated to the work file. Since blocks are never deallocated, this variable is only incremented, never decremented.

Inserting and Deleting Lines

We have just seen how to insert and delete entire blocks. Let us now take a closer look *at the format of each* disk block (see Figure 5, page 39). A block consists of an 8-byte header field followed by a data field. The header contains the two link fields, d_next and d_back, which we have already discussed. The header also contains two other fields. The d_lines field is a count of the number of lines contained in the data field. The d_avail field is the number of unused bytes in the block.

Lines are not split across blocks; if a block is not big enough to hold another line, one or more lines are copied to a new block. Almost all blocks contain a few unused bytes, but this is not a great disadvantage. (The average amount of waste in a block is one half of the average size of a line.) The number of lines in the data field is indicated by the d_lines field. Each line is preceded by a two-byte count of the number of characters in the line. No end-of-line character is used in the work file; thus, the net overhead due to the count field is just one byte.

To delete a line the data area is compressed to eliminate the line, the d_lines field is decremented, and the d_avail field is increased by the length of the line. Similarly, to insert a line a hole is made in the data area for the new line, the new line is copied into the hole, and the header (the d_lines and the d_avail fields) is adjusted to reflect the change.

Whenever a line is deleted, a check is made of the neighboring blocks to see whether any blocks can be merged. Merging is very important as it drastically reduces wasted space in the work file. Two blocks are merged by copying the data field of the second block to the available area of the first block. The header of the first block is adjusted and the second block is deleted. This is illustrated in Figures 6a and 6b (page 39).

Things get a bit complicated if there is not enough room in the block for an added line. There are three cases (see Figures 7a, 7b and 7c, page 43), depending on how big the new line is and where in the block the new line goes. **Case 1**: If the new line will fit in the original block, but the following lines will not, then a new block is created for the following lines. **Case 2**: If the new line will not fit in the original block, a new block is created for it. If the following lines fit in this new block, they also are copied to the new block. **Case 3**: If the following lines do not fit in the new block, a *second* new block is created just for them.

Replacing one line by another is done simply by deleting the old line and inserting the new line. This may seem a bit slow, but the delay is not noticeable. More importantly, using the delete routine in this way insures that blocks will be merged whenever possible. This crude approach to replacing lines saves a lot of code.

Moving through the File

How can the buffer routines find the start of a particular line in the work file? This creates surprising problems. When I first designed the buffer routines, I planned to use an *index*, an array of pointers to the start of each line. I did not want to limit the index to what would fit into memory since the size of the index would then limit the size of the file that could be edited. (Many editors which purport to be able to handle large files actually suffer this limitation including, for example, the editors derived from the book *Software Tools.*) I considered putting at least part of the index on a file. At first this seemed feasible, but in fact it is not. The reason is that the index must be accessed just like an array: given a line, we want to find the pointer to the line *without searching.* But making insertions or deletions to an array requires moving *all* entries in the array which follow the inserted or deleted entry. This is much too expensive if the array is held on a disk file.

I also investigated using the kinds of data structures common in data base management systems. An example is the B-Tree. (See "The Ubiquitous B-Tree," *ACM Computing Surveys,* Volume 11, Number 2, June 1979.) The basic problem with all these structures is that they assume that the key associated with a data item is fixed. But in the case of an editor, a data item is a line of text and the associated key is the line number. The line number changes whenever an insertion or deletion is made to the work file. In short, I can find no way to adapt B-Trees, or any similar structure, to the problem of creating an index for a text editor. If anyone knows of a way, I would love to hear about it. (Aside: in some editors the line number of a line does *not* change when insertions or deletions are made. Such editors can, and do, use index files.)

With no index available, the buffer

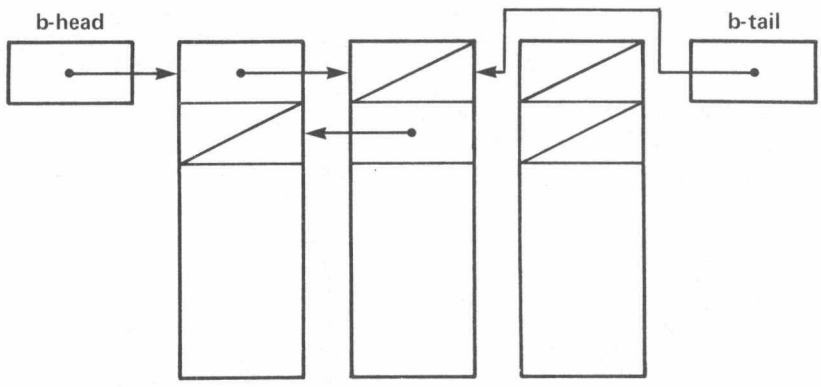


Figure 3a.

Inserting or deleting a block at the end of the work file is a special case.

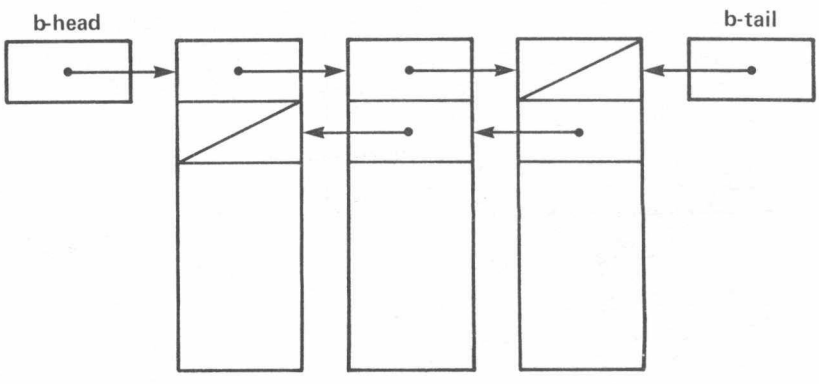


Figure 3b.

routines must search through the file block by block. Surprisingly enough, this search is almost always very fast! There are three reasons for this:

First, the requested line is usually very near the current line. In fact, most often the requested line is either the immediately preceding or following line. For example, when scrolling the screen or when using the "find" command, the editor steps through the file line by line.

Second, when the requested line is far away from the current line the new line is most often near either the beginning or end of the file. The beginning and ending blocks of the work file can be accessed without searching because the editor always keeps track of where they are. (This is the reason for the b_max_line and b_tail variables.)

Third, each block contains a count of the number of lines in the block. Thus, it is simple to tell whether a block contains the desired line without actually searching the data field of the block.

Virtual Memory and The LRU Algorithm

Up until now I have been tacitly assuming that all blocks have been loaded into memory when needed. In other words, the main memory appears to be of unlimited size. Since the work file may grow to be much larger than the available main memory, blocks must be swapped into and out of memory as needed. Here is how virtual memory works:

The buffer routines contain a set of *slots,* each of which contains one block and some status information which is not saved on the disk. This status information tells whether the slot is free or occupied, where on the disk the block is stored, and other information which we will discuss a little later.

Blocks are loaded into a slot only by explicitly calling the swap_in() routine. Swap_in() first checks to see whether the requested block already resides in a slot. If so, swap_in() simply returns the slot's number. Otherwise, swap_in() searches for an empty slot. If one is found, the requested block is loaded into the empty slot.

If no empty slot is found, a slot must be swapped out to the disk. The slot which is chosen is the Least Recently Used (LRU) slot. Every time a slot is accessed (i.e., every time swap_in() is called), the d_lru field in the status table is adjusted in such a way that the least recently used slot has the highest value. When a slot must be swapped out, the slot with the largest d_lru field is chosen.

Once the slot has been chosen, the swap_in() routine checks the status table to see if the block has been changed since it was loaded into the slot. This information is popularly called the *dirty bit,* which is part of the d_status field. The dirty bit is cleared when a block is first loaded into a slot, and it is set whenever the block is changed. If the dirty bit is set, the block must be swapped out to the disk before the new block replaces it. Otherwise, the new block can be loaded in over the old block because the disk still contains a correct copy of the old block.

Using a dirty bit complicates the buffer routines quite a bit. For instance, failing to set the dirty bit properly creates a "time bomb." The code appears to work correctly, but the disk contains an incorrect copy of a block. The error only shows up when (possibly much later) the block is reloaded into memory. Tracy Kidder discusses this problem in a different context in his Pulitzer Prize winning book, *The Soul of a New Machine.* (If you haven't yet read it, drop everything and do so at once!) In spite of what I have just said about the dirty bit, using it is *essential.* Without the dirty bit, many more disk accesses will be needed.

In practice, the LRU algorithm works well. It eliminates disk accesses when editing in a localized area, which is most of the time. It also performs reasonably well when scrolling through the file.

It is instructive to compare the performance of the LRU algorithm with a simpler approach. For example, Figure 8 (page 44) shows one of my first ideas for creating a virtual memory. Briefly, a memory buffer contains just a few lines. All lines before these lines are held in a *head file;* all lines after the lines in the buffer are held in a *tail file.* The advantage of this scheme is that all disk activity is to the physical end of each file. This method can be programmed even on systems which do not support random access to files. Also, lines are changed only in the main buffer; there is no need to merge or split blocks.

Using two files in this way, however, is flawed. Moving through the file requires that lines be moved through the memory buffer and written to one of the two files. For instance, moving to the start of the file requires that most of the file be written to the tail file. A little thought should convince you that the performance of this method is essentially the same as using the LRU algorithm *without* the dirty bit.

Performance

N+1 trivial tasks are expected to be accomplished in the same time as N trivial tasks.
– Gray's Law of Programming

N + 1 trivial tasks take twice as long as N trivial tasks for N sufficiently large.
– Logg's Rebuttal to Gray's Law

In this section, I'd like to compare the speed of the buffer routines used in ED2 and RED. It is important to realize that you may tune the performance of RED's buffer routines by changing some compile time-constants defined on file red 10.sc (see Listing, page 50). The speed-up may be dramatic, depending on your disk drives.

In general, RED will run faster the more buffers there are and the larger each buffer is. You change the *size* of the buffers by changing the DATA_SIZE constant. You change the *number* of buffers by changing the NSLOTS constant. If you have room in memory, I recommend setting DATA_SIZE to 1024 and NSLOTS to 4 (or more).

Reading a file into the buffer is always considerably slower with these new routines because the file being edited must be copied to the work file. The important fact is that large files are loaded at the same *rate* as small files.

Scrolling the screen will be interrupted from time to time when the LRU algorithm swaps blocks in and out. On a floppy disk this delay is slightly annoying but tolerable. On a hard disk the delay is barely noticeable. In any event, the delay does not depend on the size of the work file, but rather on the speed of the disk drive, the block size, and the position of the work file. (The closer the work file is to the directory, the better.)

Inserting or replacing lines is extremely fast, unless the operation results in a block being swapped in or out. For large files, these operations are *much faster* than with the old buffer routines.

Deleting many lines can take longer with these routines because the deleted blocks must still be written to the disk in order to keep the free list up to date.

Keep in mind that splitting or joining blocks usually does not cause blocks to be swapped from the disk because the LRU algorithm increases the odds that those blocks are already in memory. These operations cause little overhead.

Recovery from Errors

The new buffer routines employ two types of error checking. Whenever a change is made to a block, a check is made of the consistency of the information in the block's header and data fields. These checks are quite effective; almost all bugs in the code were caught this way. The checks have even pointed out a bad memory chip. You can disable these checks by commenting out all the calls to check_block() *on file red12.sc.* I don't recommend that you do this because the checks do not slow down the editor at all. On the other hand, if you eliminate all calls to check_block() you don't have to #include the file red13.sc. This will save you some space if you really need it.

The other type of error checking is reported by the operating system to the sysseek(), sysread(), and syswrite() routines. Recovering from these errors is a

difficult problem. Ideally, the editor could keep enough information around to "back out" of any transaction that cannot be completed, but this cannot be done unless a truly enormous amount of information is passed between the routines that detect errors and routines that might do something about the errors. The following paragraphs discuss the three main types of disk error and how RED recovers.

(1) The disk becomes full. This is by far the most likely cause of a disk error. The error is reported on the screen and the operation which caused the error is aborted. The buffer is cleared if the load command caused the error.

(2) The disk is removed during editing. This creates a real problem because CP/M completely botches the error recovery. CP/M gives the message, "BDOS error on A: select," then proceeds to mark the disk as read only! If a write is in progress, CP/M will then abort the editor without ever returning control. So *never* remove a disk while RED is running.

If this kind of error should happen to you, remember that the disk still contains a copy of the work file. To recover the information from the work file do the following:

- Rename the work file so that you don't erase it by invoking the editor!
- Write a short *recovery program*. This program should be patterned after the write_file() routine in file red12.sc. The recovery program should copy the work file to another file deleting header information as it goes. Note that block 0 of the work file *always* contains the first block of the file, but the following blocks may be anywhere in the work file. Use the d_next field in the header of each block to tell where the next block is to be found.

(3) There is a "real" disk error. In other words, the hardware that controls the disk fails to read or to write a sector properly. Some case could be made for retrying the read or write operation a few times to see if the problem would go away. In my experience, this kind of problem is very rare, so the buffer routines don't attempt any fancy recovery. Besides, accessing the disk further might just compound the problem. The best thing to do is to stop editing and start figuring out why the disk error has occurred.

In short, the new buffer routines are probably more reliable than the old routines. Even in an extreme emergency, the work file may be used to recover your work.

In the ED2 editor, errors were indicated by returning a status value from all buffer routines. In RED, this is done by setting three disk restart points – one for each mode that the editor can be in. When a disk error occurs, the longjmp() function (defined in the BDS C library) is used to perform a jump to the current restart point. Here is an example of a *goto* instruction being justified on the basis of eliminating a lot of extraneous code.

Summary

To summarize the important features of RED's new buffer routines:

- **Huge files may be edited** because main memory contains no data structures whose size depends on the size of the file. In addition, the work file never contains much garbage – typically, the work file is no more than 10% larger than the original file being edited. Files are limited in size only by (1) how much room exists on the disk for the work file and (2) how large the operating system allows a file to be.
- **The response time for editing is constant.** Inserting, deleting, or replacing a line takes the same time regardless of the length of the file being edited. I have edited files of 20,000 lines easily.
- **The response time for file operations is linear.** Reading, writing, and appending files proceed at a constant rate, regardless of the size of the files. In other words, reading a 10,000 line program takes only 10 times longer than reading a 1,000 line program.

Next month I'll discuss the Small C library used in RED and improvements that might be made to RED. I'll also say a few words about why RED is copyrighted.

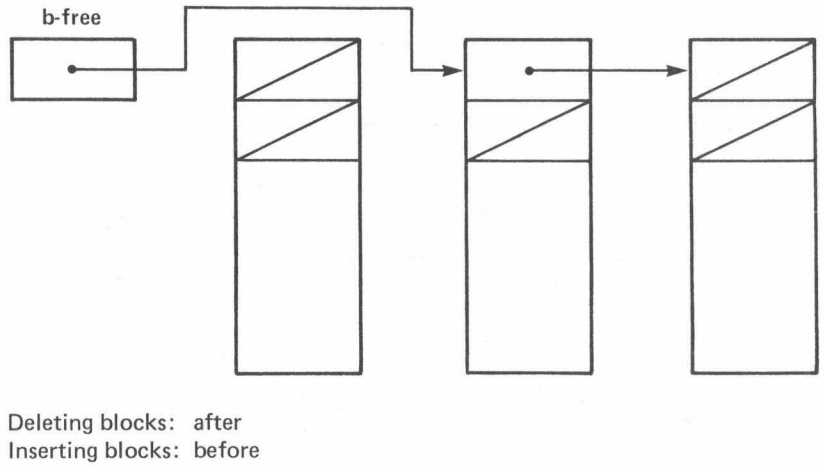


Figure 4a.

All insertions and deletions are made at the head of the free list.

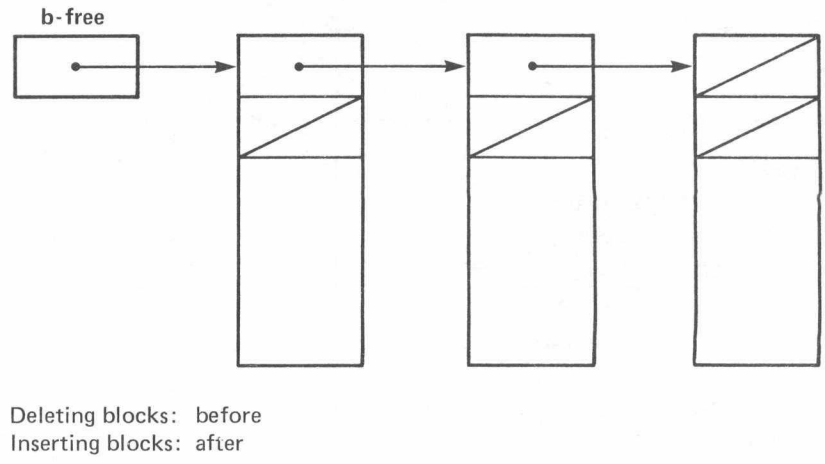


Figure 4b.

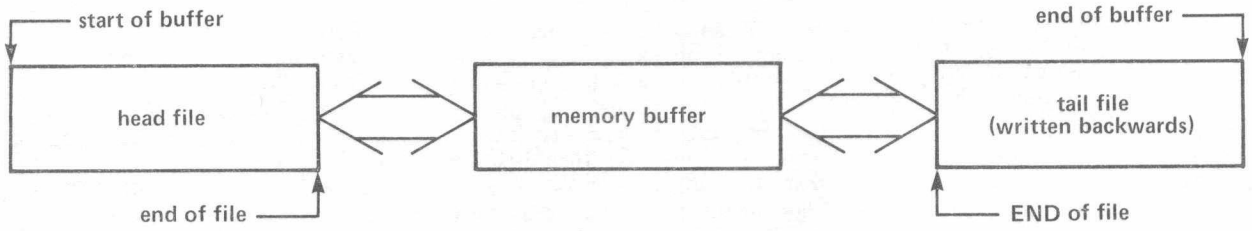


Figure 8.

An alternative way of providing a huge buffer. This method causes many extra disk accesses.

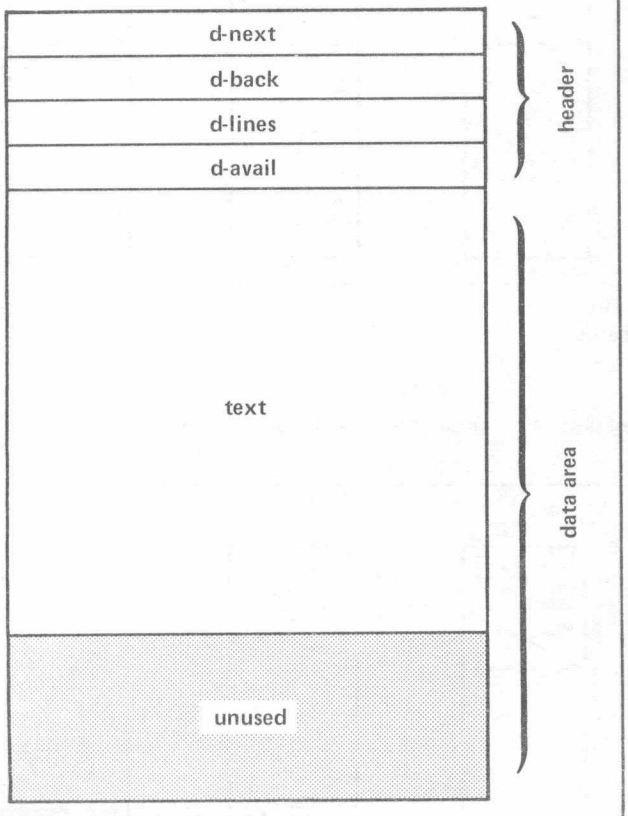


Figure 5.

A block consists of a header field, which tells what is stored in the block, and the data field, which contains lines of text.

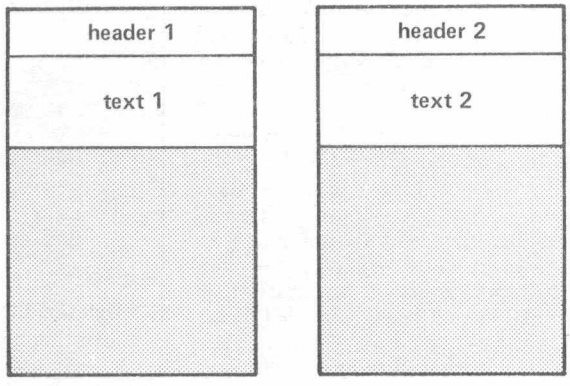


Figure 6a.

Two blocks before they are merged.

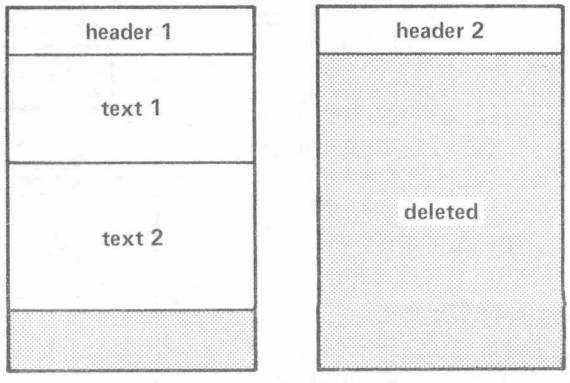
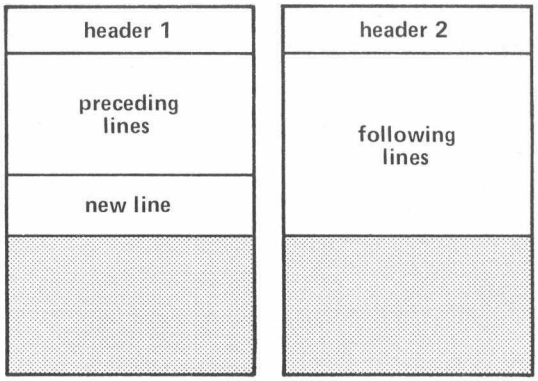
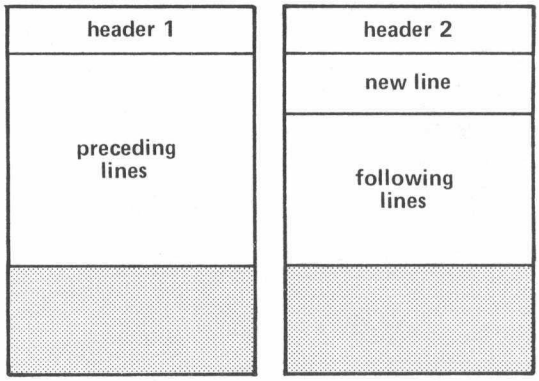


Figure 6b.

Two blocks after merging. Lines are copied from the second block, the header of the first block is adjusted and the second block is deleted.





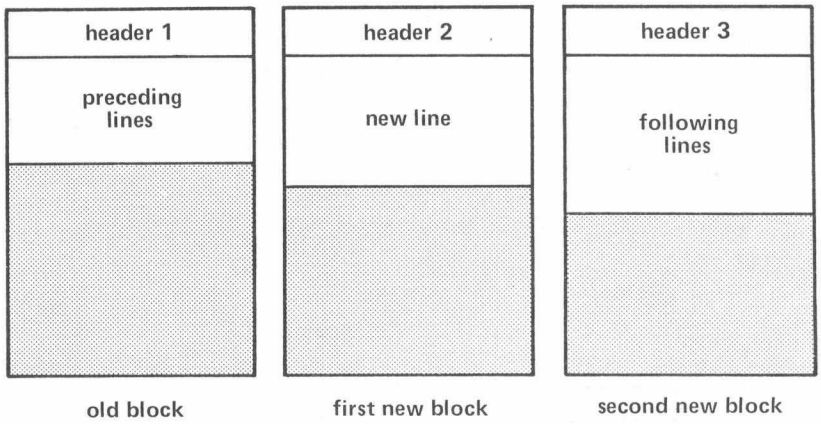


Figure 7.

In (a), the new line fits into the old block; (b) shows both the new line and the following lines fitting into the new block; and (c) shows two new blocks created.

RED - Listing

(Text begins on page 34)

```
/*
        Include file for RED:  small-C version

        Source:  red.sc
        Version: January 20, 1983

        Copyright (C) 1983 by  Edward K. Ream
*/

#include a:red0.sc       /* constants, (system) globals  */
#include a:red1.sc       /* user defined globals         */
#include a:red2.sc       /* main program                 */
#include a:red3.sc       /* command mode commands        */
#include a:red4.sc       /* edit mode module             */
#include a:red5.sc       /* output format module         */
#include a:red6.sc       /* display screen module        */
#include a:red7.sc       /* prompt line module           */
#include a:red8.sc       /* operating system module      */
#include a:red9.sc       /* utility routines             */
#include a:red10.sc      /* buffer module -- part 1      */
#include a:red11.sc      /*                -- part 2     */
#include a:red12.sc      /*                -- part 3     */
#include a:red13.sc      /* debugging routines           */
#include a:lib1          /* machine library              */
#include a:lib2          /* init, exit, utility library  */
#include a:fiolib1       /* file library                 */
#include a:xlib          /* setjmp, longjmp              */
```

```
/*
        RED's non-user defined globals -- small-C version

        Source:  red0.sc
        Version: see below

        Copyright (C) 1983 by Edward K. Ream
*/


/* Define global constants */

#define SIGNON  "Welcome to RED"
#define VERSION "Small-C Version 4.04,  February 27, 1983"
#define COPYRIGHT "Copyright (C) 1983 by Edward K. Ream"
#define XSIGN   "Type 'help' in command mode for help"
#define XSIGN1 "Type 'h' in edit mode for more help"

/* Define constants describing a text line */

#define MAXLEN  200     /* max chars per line           */
#define MAXLEN1 201     /* MAXLEN + 1                   */

/* Define operating system constants */

#define SYSFNMAX 15     /* CP/M file name length + 1    */
#define CPMEOF 26

/* Define misc. constants */

#define EOS     0       /* code sometimes assumes \0    */
#define OK      1
#define ERR     -1      /* error.  must be <0           */
#define ERROR   -1      /* error.  must be <0           */
#define EOF     -2      /* end of file.  must be <0     */
#define YES     1       /* must be nonzero              */
#define NO      0
#define CR      13      /* carriage return              */
#define LF      10      /* line feed                    */
#define TAB     9       /* tab character                */
#define HUGE    32000   /* practical infinity           */
```

```
/*
RED special key definitions
Source:  red1.sc
This file was created by the configuration program:
January 20, 1983.
*/

/*
Define which keys are used for special edit functions.
*/

#define UP1 10
#define DOWN1 13
#define UP2 21
#define DOWN2 4
#define LEFT1 12
#define RIGHT1 18
#define INS1 14
#define EDIT1 5
#define ESC1 27
#define DEL1 8
#define ZAP1 26
#define ABT1 24
#define SPLT1 19
#define JOIN1 16
#define REP1 1

/*
Define length and width of screen and printer.
*/

#define SCRNW 80
#define SCRNW1 79
#define SCRNL 24
#define SCRNL1 23
#define SCRNL2 22
#define LISTW 132
```

```
/*
        RED operating system module -- small-C version

        Source:  red8.sc
        Version: January 23, 1983

        Copyright (C) 1983 by Edward K. Ream
*/


/* Define globals used by this module */

char sysinbuf[128];     /* file buffer */
int  sysincnt;          /* buffer count */

char syscbuf[MAXLEN];   /* console type ahead buffer */
int  sysccnt;

int sysrcnt;            /* repeat count */
int syslastc;           /* last character (may be repeated) */

int  systopl,systopy,sysnl;     /* interrupt information */

#define READ_SIZE 4     /* see also definition in red10.sc */


/*      Initialize the system module. */

sysinit()
{
        sysnl    = 0;
        sysccnt  = 0;
        sysrcnt  = 0;
        syslastc = 0;
}


/*      Save info for interrupted screen update. */

sysintr(systl, systy, sysn)
int systl, systy, sysn;
{
        systopl = systl;
        systopy = systy;
        sysnl   = max(0,sysn);
}


/*
        Return -1 if no character is ready from the keyboard.
        Otherwise, return the character.

        This routine handles typeahead and the repeat key.
*/

syscstat()
{
        int c, i;

        /* Always look for another character. */
        c = bdos(6,-1);

        if ( (c == REP1) & (syslastc != 0) ) {
                sysrcnt = max(1, 2*sysrcnt);


                i = 0;
                while ( (i++ < sysrcnt) & (sysccnt < MAXLEN) ) {
                        syscbuf [sysccnt++] = syslastc;
                }
        }
        else if (c != 0) {
                syslastc = c;
                sysrcnt  = 0;
                syscbuf [sysccnt++] = c;
        }

        if (sysccnt > 0) {
                return syscbuf [--sysccnt];
        }
        else {
                return -1;
        }
}
```

(Continued on page 48)

```
/*
        Wait for next character from the console.
        Do not echo it.
        This routine prints any waiting lines if there is no
*/      input ready

syscin()
{
        int c;

        while ((c=syscstat()) == -1) {

                /* Output queued ? */
                if (sysnl > 0) {
                        bufout(systopl, systopy, sysnl);
                }
        }
        return c;
}


/*
        Wait for all console output to be finished.
*/

syswait()
{
        while (sysnl > 0) {
                bufout(systopl, systopy, sysnl);
        }
}


/* Print character on the console. */

syscout(c)
char c;
{
        bdos(6,c);
        return(c);
}


/* Print one character on the printer. */

syslout(c) char c;


{
        bdos(5,c);
        return(c);
}


/* Close a file which was opened by sysopen() or syscreat(). */

sysclose(file)
int file;
{
        return close(file);
}


/*
        Create a file.  Erase it if it exists.
        Leave the file open for read/write access.
*/

syscreat(filename)
char * filename;
{
        return creat(filename);
}


/* Return YES if the file exists. */

sysexists(filename)
char * filename;
{
        int file;

        if ((file = open(filename, 0)) != ERROR) {
                close(file);
                return YES;
        }
        else {
                return NO;
        }
}


/*
        Open a file which already exists.
        Mode 0 -- read only.
        Mode 1 -- write only.
        Mode 2 -- read/write.
*/

sysopen(name, mode)
char *name;
int mode;
{
        /* Kludge:  set count for sysgetc(). */
        if (mode == 0) {
                sysincnt = 128;
        }

        return open(name, mode);
}


/*
        Read next line from a file.

        End the line with a zero byte.
        Only one file at a time may use this routine.
*/

sysrdln(file, buffer, maxlength)
int file;
char *buffer;
int maxlength;
{
        int c, count;

        count = 0;
        while(1) {
                c = sysgetc(file);
                if (c == CR) {
                        continue;
                }
                else if (c == CPMEOF) {
                        buffer [count = EOS];
                        return ERROR;
                }
                else if (c == LF) {
                        buffer [count] = EOS;
                        return count;
                }
                else if (count < maxlength - 1) {
                        buffer [count++] = c;
                }
                else {
                        count++;
                }
        }
}


/*
        Get one character from the input file.
        Only one file at a time may use this routine.
*/


sysgetc(file)
int file;
{
        int n;

        if (sysincnt == 128) {
                n = read(file, sysinbuf, 1);
                if (n == ERROR) {
                        disk_error("read error");
                        return CPMEOF;
                }
                else if (n == 0) {
                        /* End of file. */
                        return CPMEOF;
                }
                else {
                        sysincnt = 0;
                }
        }
        return sysinbuf [sysincnt++];
}


/*
        Read one block (READ_SIZE sectors) into the buffer.
*/

sysread(file, buffer)
int file;
char * buffer;
{
        return read(file, buffer, READ_SIZE);
}


/*      Write n sectors from the buffer to the file. */

syswrite(file, buffer, n)
int file;
char * buffer;
int n;
```

(Continued on page 50)

RED - Listing

(Listing continued, text begins on page 34)

```
{
        return write(file, buffer, n);
}


/* Seek to a specified block of an open file. */

sysseek(file, block)
int file, block;
{
        return seek(file, block * READ_SIZE, 0);
}


/* Remove the file from the file system. */

sysunlink(filename)
char * filename;
{
        return unlink(filename);
}


/* Check file name for syntax. */

syschkfn(args) char *args;
{
        return(OK);
}


/* Copy file name from args to buffer. */

syscopfn(args,buffer) char *args, *buffer;
{
int n;
        n=0;
        while (n<(SYSFNMAX-1)) {
                if (args[n]==EOS) {
                        break;
                }
                else {
                        buffer[n]=args[n];
                        n++;
                }
        }
        buffer[n]=EOS;
}
```

```
/*
        RED buffer routines -- small-C version
        Part 1 -- goto, output and status routines.

        Source:  red10.sc
        Version: August 28, 1982; January 23, 1983

        Copyright (C) 1983 by Edward K. Ream
*/


/*
        You may tune these constants for better performance.

        DATA_SIZE:  The size of struct BLOCK.
                    Make sure that DATA_SIZE is a multiple
                    of the size of your disk sectors.
                    (for CP/M, a multiple of 128)
                    Make sure that the READ_SIZE constant in
                    ed8.c is DATA_SIZE / 128.

        NSLOTS:     The number of BLOCKS resident in memory.
                    The code assumes this number is AT LEAST 3.

        DATA_FILE:  The name of the work file. Note the double
                    quotes.  Pick a name you never use.
*/

#define DATA_SIZE 512
#define NSLOTS 5
#define DATA_FILE "@@data@@.tmp"


/* Do not touch this constant. */

#define CPM_SIZE 128


int     b_fatal;        /* erase buffer on disk error   */
int     b_cflag;        /* buffer changed flag          */

int     b_line;         /* current line number          */
int     b_max_line;     /* highest line number          */
char *  b_linep;        /* pointer to line (local var)  */

int     b_slot;         /* current block's slot number  */
int     b_start;        /* first line of current block  */
int     b_head;         /* first block's disk pointer   */
int     b_tail;         /* last block's disk pointer    */
int     b_max_diskp;    /* last sector allocated        */

int     b_data_fd;      /* file descriptor of data file */
int     b_user_fd;      /* file descriptor of user file */
int     b_free;         /* head of list of free blocks  */

char    b_buff [DATA_SIZE];     /* temporary buffer.    */


/*
        Partially define the format of a block.  The data
        field is organized as a singly linked list of lines;
        that is, each line is preceded by a two-byte length
        field.
        The d_back and d_next fields in the header are used
        to doubly-link the disk blocks so that stepping
        through the blocks either forward or backwards is
        efficient.  -1 denotes the end of each list.

        When blocks become totally empty they are entered
        on a list of free blocks.  The links of this list
        are kept in the blocks themselves in the d_next field.
        The b_free variable is the head of this list.

        Also define the in-core block table.  This table
        contains the blocks that have been swapped into
        memory.  Each entry in this table is called a slot.
*/

/* comment out ----------

struct BLOCK {
        int     d_back;
        int     d_next;
        int     d_avail;
        int     d_lines;
        char    d_data [BUFF_SIZE];
}d_blocks [NSLOTS];

---------- end comment out */


#define HEADER_SIZE 8           /* size of first 4 fields  */
#define BUFF_SIZE 504           /* DATA_SIZE - HEADER_SIZE */
#define SLOT_SIZE 2560          /* NSLOTS * DATA_SIZE      */


int  d_back  [NSLOTS];          /* # of previous block  */
int  d_next  [NSLOTS];          /* # of next block      */
int  d_avail [NSLOTS];          /* # of data bytes free */
int  d_lines [NSLOTS];          /* # of lines on block  */
char d_data  [SLOT_SIZE];       /* resident blocks      */



/*
        Define the resident status table.
        There is one entry for each slot.
*/

#define FREE    1       /* status:  block is available  */
#define FULL    2       /* status:  block is allocated  */
#define DIRTY   3       /* status:  must swap out       */

/* comment out ----------

struct STATUS {
        int     d_lru;
        int     d_status;
        int     d_diskp;
} d_stat_tab [NSLOTS];

---------- end comment out */

int     d_lru    [NSLOTS];      /* lru count            */
int     d_status [NSLOTS];      /* FULL, FREE or DIRTY  */
int     d_diskp  [NSLOTS];      /* disk pointer         */


/*


        Boundary conditions:

        1.  Only bufins() can extend the buffer, NOT
            bufgo() and bufdn().

        2.  bufatbot() is true when the current line is
            PASSED the last line of the buffer.  Both
            bufgo() and bufdn() can cause bufatbot() to
            become true.  bufgetln() returns a zero length
            line if bufatbot() is true.

        3.  b_max_line is the number of lines in the buffer.
            However, b_line == b_max_line + 1 is valid and
            it means that b_line points at a null line.
```

```
        4.  All buffer routines assume that the variables
            b_slot, b_line and b_start describe the
            current line when the routine is called.  Thus,
            any routine which changes the current line must
            update these variables.
*/


/* Return YES if at bottom of buffer (past the last line). */

bufatbot()
{
        return (b_line > b_max_line);
}


/* Return YES if at top of buffer. */

bufattop()
{
        return (b_line == 1);
}


/* Return YES if the buffer has been changed. */

bufchng()
{
        return b_cflag;
}


/* Move towards end of buffer. */

bufdn()
{
        /* The call to buf_gofw() instead of bufgo()
         * is made purely to increase speed slightly.
         */
        if (bufatbot()) {
                return;
        }
        else {
                b_line++;
                buf_gofw();
        }
}


/* Clean up any temporary files. */

bufend()
{
        sysunlink(DATA_FILE);
}


/*
        Go to line n.
        Set b_slot, b_line, b_start.
*/

bufgo(n)
int n;
{
        int distance, oldline;

        /* Put the request in range. */
        oldline = b_line;
        b_line = min (n, b_max_line + 1);
        b_line = max (1, b_line);
        distance = b_line - oldline;

        if (distance == 0) {

                /* We are already at the requested line. */
                return;

        }
        else if (distance == 1) {

                /* Go forward from here. */
                buf_gofw();
                return;
        }
        else if (distance == -1) {

                /* Go back from here. */
                buf_gobk();
                return;

        }
        else if (distance > 0) {
                if ( b_line >
                     oldline + ((b_max_line - oldline) / 2)
                   ) {
                        /* Search back from end of file. */
                        swap_in(b_tail, &b_slot);
                        b_start =
                            1 + b_max_line - d_lines [b_slot];
                        buf_gobk();
                        return;
                }
                else {

                        /* Search forward from here. */
                        buf_gofw();
                        return;
                }
        }
        else {
                if (b_line < oldline / 2) {

                        /* Search from start of file. */
                        swap_in(b_head, &b_slot);
                        b_start = 1;
                        buf_gofw();
                        return;

                }
                else {

                        /* Search back from here. */
                        buf_gobk();
                        return;
                }
        }
}


/*
        Search backwards from block for b_line.
        The starting line number of the block is b_start.
        Set b_slot and b_start.
 */

buf_gobk ()
{
        int diskp;

        if ( (b_slot == ERROR) |
             (b_start < 1)     | (b_start > b_max_line) |
             (b_line  < 1)     | (b_line  > b_max_line + 1)
           ) {

                cant_happen("buf_gobk 1");
        }

        /* Scan backward for the proper block. */
        while (b_start > b_line) {

                /* Get the previous block in memory. */
                diskp = d_back [b_slot];
                if (diskp == ERROR) {
                        cant_happen("buf_gobk 2");
                }
                swap_in(diskp, &b_slot);

                /* Calculate the start of the next block. */
                b_start = b_start - d_lines [b_slot];
                if (b_start <= 0) {
                        cant_happen("buf_gobk 3");
                }
        }
}


/*
        Search forward for line n.
        Set b_slot and b_start.
 */

buf_gofw ()
{
        int diskp;

        /* The last line is always null. */
        if (bufatbot()) {
                return;
        }

        if ( (b_slot == ERROR) | (b_start < b_start) |
             (b_start < 1)     | (b_start > b_max_line) |
             (b_line  < 1)     | (b_line  > b_max_line + 1)
           ) {

                cant_happen("buf_gofw 1");
        }

        /* Scan forward to the proper block. */
        while (b_start + d_lines [b_slot] <= b_line) {

                /* Get the start of the next block. */
                b_start = b_start + d_lines [b_slot];

                /* Swap in the next block. */
                diskp = d_next [b_slot];
                if ((diskp == ERROR) | (b_start > b_max_line)){
                        cant_happen("buf_gofw 2");
                }
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 34)

```
                swap_in(diskp, &b_slot);
        }
}


/* Return the current line number. */

bufln()
{
        return b_line;
}


/* Initialize the buffer module. */

bufnew()
{
        int i;
        char * static;

        /*
                Initialize b_data_fd on the first call
                to this routine.  A kludge is required
                because small-C has neither real static
                variables nor initializers.
        */


        static = "";
        if (*static == 0) {
                *static = 1;
                b_data_fd = ERROR;
        }

        /* The free list is empty. */
        b_free = ERROR;

        /* Free all slots. */
        i = 0;

        while (i < NSLOTS) {
                d_status [i] = FREE;
                d_lru    [i] = i;
                d_diskp  [i] = ERROR;
                i++;
        }
        /* Allocate the first slot. */
        b_slot = b_head = b_tail = 0;
        b_max_disp = 1;
        d_diskp  [b_slot] = 0;
        d_status [b_slot] = DIRTY;
        d_back   [b_slot] = d_next [b_slot] = ERROR;

        /* The first slot is empty. */
        d_lines [b_slot] = 0;
        d_avail [b_slot] = BUFF_SIZE;

        /* Make sure temp file is erased. */
        if (b_data_fd != ERROR) {
                sysclose(b_data_fd);
                b_data_fd = ERROR;
                sysunlink(DATA_FILE);
        }

        /* Set the current and last line counts. */
        b_line = 1;
        b_max_line = 0;
        b_start = 1;

        /* Indicate that the buffer has not been changed. */
        b_cflag = NO;

        /* Do not erase the work file on a disk error. */
        b_fatal = NO;
}


/* Return YES if buffer is near the bottom line */

bufnrbot()
{
        return (b_line >= b_max_line);
}
/* Put nlines lines from buffer starting with line topline at
 * position topy of the screen.
 */

b[illegible](topline, topy, nlines)
[illegible] topline, topy, nlines;

{
        int i, x, y;

        x = outgetx();
        y = outgety();
        l = b_line;
        while (nlines > 0) {
                outxy(0, topy++);
                bufoutln(topline++);
                nlines--;
                sysintr(topline, topy, nlines);
                break;
        }
        outxy(x,y);
        bufgo(l);
}


/* Print one line on screen. */

bufoutln(line)
int line;
{
        char buffer [MAXLEN1];
        int n;

        bufgo(line);

        if (bufatbot()) {
                outdeol();
        }
        else {
                n = bufgetln(buffer, MAXLEN);
                n = min(n, MAXLEN);
                buffer [n] = CR;
                fmtsout(buffer, 0);
                outdeol();
        }
}



/* Replace current line with the line that p points to.
 * The new line is of length n.
 */

bufrepl(line, n)
char line [];
int n;
{
        /* Do not replace null line.  Just insert. */
        if (bufatbot()) {
                bufins(line, n);
                return;
        }
        bufdel();
        bufins(line, n);
}


/* Indicate that the file has been saved. */

bufsaved()
{
        b_cflag = NO;
}


/* Move towards the head of the file. */

bufup()
{
        /* The call to buf_gobk() instead of buf_go()
         * is made purely to increase speed slightly.
         */

        if (bufattop()) {
                return;
        }
        else {
                b_line--;
                buf_gobk();
        }
}
```

```
/*
        RED buffer routines -- small-C version
        Part 2 -- line routines.

        Source:  red11.sc
        Version: August 7, 1982; February 26, 1983

        Copyright (C) 1983 by Edward K. Ream
*/


/*

        Scan for the start of the current line.
        Return *count  = the # of characters in the line.
        Return *prefix = the # of characters before the line.
*/

b_scan(pointer, count, prefix)
int     *pointer;       /* Kludge -- should be char ** */
int     *count;
int     *prefix;
{
        char    *cp;
        int     i, limit, count1;
```

```
        /* The last line is always null. */
        if (bufatbot()) {
                *prefix  = BUFF_SIZE - d_avail [b_slot];
                *pointer = data_addr(b_slot) + *prefix;
                *count   = 0;
                return;
        }

        /* Limit is the starting line # of the next block. */
        limit = b_start + d_lines [b_slot];

        /* Point pointer at the start of the first line. */
        cp = data_addr (b_slot);

        /* Keep track of characters before the line. */
        *prefix = 0;

        i = b_start;
        while (1) {

                if (i == limit) {
                        cant_happen("b_scan 1");
                }

                /* Get length of the line. */
                count1   = b_getnum(cp);

                if ( (count1 < 0) | (*prefix >= BUFF_SIZE) ) {
                        cant_happen("b_scan 2");
                }

                /* At the requested line? */
                if (i == b_line) {
                        break;
                }

                /* Step over the count and the line. */
                cp      = cp + count1 + 2;
                *prefix = *prefix + count1 + 2;
                i++;
        }

        /* Point past the line length. */
        cp = cp + 2;

        /* Set values in the calling routines. */
        *pointer = cp;
        *count = count1;
}

/*
        Get and put a 2-byte line number.
        These are machine independent substitutes for casts:
*/

b_getnum(cp)
char *cp;
{
        /*      simulate:  ip = (*int) cp; return *ip;  */

        return (*cp << 8) | *(cp+1);
}

b_putnum(cp, num)
char *cp;
int num;
{
        /*      simulate:  ip = (*int) cp; *ip = num;   */

        *cp     = (num & (255 << 8)) >> 8;
        *(cp+1) = num & 255;
}


/* Delete the current line. */

bufdel()
{
        char    *p, *p1, *q;
        int     length, junk;
        int     endp;
        int     back, current, next;

        /* Do nothing if the buffer is empty. */
        if (bufatbot()) {
                return;
        }

        /* The current block will become dirty. */
        is_dirty(b_slot);
        b_cflag = YES;

        /*
                Point p  at the line # of the deleted line.
                Point p1 at the line # of the following line.
                Point q  passed the last byte of the block.
        */

        b_scan(&p, &length, &junk);
        endp = BUFF_SIZE - d_avail [b_slot];
        q  = data_addr (b_slot) + endp;
        p1 = p + length;
        p  = p - 2;

        /* Compress the block. */
        while (p1 != q) {
                *p++ = *p1++;
        }

        /* Adjust the avail and line counts in the block. */
        d_avail [b_slot] = d_avail [b_slot] + length + 2;
        d_lines [b_slot]--;

        /* Decrease the overall line count. */
        b_max_line--;

        /* Point to the previous, current and next blocks. */
        back    = d_back  [b_slot];
        current = d_diskp [b_slot];
        next    = d_next  [b_slot];

        /* Move to the correct block. */

        if ( (next == ERROR) & (d_lines [b_slot] == 0) ) {
                /* The last block is empty.  Move back. */
                if (back != ERROR) {
                        swap_in(back, &b_slot);
                        b_start = b_max_line -
                                        d_lines [b_slot] + 1;
                }
        }

        else if (b_start + d_lines [b_slot] == b_line) {
                /* The line moves to the next block. */
                if (next != ERROR) {
                        swap_in(next, &b_slot);
                        b_start = b_line;
                }
        }

        /*
                Combine blocks if possible.
                This is tricky code because combine() causes
                side effects.  Do not try to pre-compute the
                arguments for the second call to combine().
        */

        combine(d_back  [b_slot], d_diskp [b_slot]);
        combine(d_diskp [b_slot], d_next  [b_slot]);


        /* ----- check code ----- */
        check_block("delete");
}


/* Delete n lines starting at the current line. */

bufdeln(n)
int n;
{
        int i;

        i = 0;
        while( (i < n) & (bufatbot() == NO) ) {
                bufdel();
                i++;
        }
}
/*
        Copy the current line from the buffer to line [].
        The size of line [] is linelen.
        Return k = the length of the line.
        If k > linelen then truncate k - linelen characters.
 */

bufgetln(line, linelen)
char    *line;
int     linelen;
{
        int     count, i, junk, limit;
        char    *cp;

        /* Return null line at the bottom of the buffer. */
        if (bufatbot()) {
                line [0] = CR;
                return 0;
        }

        /* Scan forward to start of the line. */
        b_scan(&cp, &count, &junk);

        /* Copy line to buffer */
        limit = min(count, linelen);
        i = 0;
        while (i < limit) {
                line [i] = cp [i];
                i++;
        }

        /* End with zero. */
        line [min (count, linelen - 1)] = EOS;
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 34)

```
        /* Return the number of characters in the line. */
        return count;
}


/*
        Insert line before the current line.  Thus, the line
        number of the current line does not change.  The line
        ends with a zero byte but not with a CR.

        This is fairly crude code, as it can end up splitting
        the current block into up to three blocks.  However,
        the combine() routine does an effective job of keeping
        the size of the average block big enough.
*/

bufins(line, linelen)
char line [];
int linelen;
{
        char    *p, *q;
        int     junk;
        int     length1;    /* # of chars before break point */
        int     length2;    /* # of chars after  break point */
        int     i;

        if (linelen > BUFF_SIZE) {
                cant_happen("bufins 1");
        }

        /* Point p at the start of the current line. */
        b_scan(&p, &junk, &length1);

        p = p - 2;

        /* Calculate # of characters after break point. */
        length2 = BUFF_SIZE - d_avail [b_slot] - length1;

        /* The current slot is now dirty. */
        is_dirty(b_slot);

        /* Allow for 2-byte line length. */
        if (length1 + linelen + 2 > BUFF_SIZE) {

                split_block(length1, length2, YES);
                length1 = 0;
                length2 = BUFF_SIZE - d_avail [b_slot];
        }

        /*
                At this point we know that the new line will
                fit on the current block at position length1.
        */

        if (linelen + 2 > d_avail [b_slot]) {

                split_block(length1, length2, NO);

                /* Copy line to the end of the old block. */
                p = data_addr (b_slot) + length1;
                i = 0;
                while (i < linelen) {
                        p [2 + i] = line [i];
                        i++;
                }

                /* Insert line length. */
                b_putnum(p, linelen);

                /* Adjust header. */
                d_avail[b_slot] = d_avail[b_slot]-linelen-2;
                d_lines[b_slot]++;
        }
        else {
                /* Make a hole in the block. */
                p = data_addr (b_slot) + length1;
                q = p + linelen + 2;

                i = length2 - 1;
                while (i >= 0) {
                        q [i] = p [i];
                        i--;
                }

                /* Put the line length in the hole. */
                b_putnum(p, linelen);

                /* Copy the new line into the hole. */
                p = p + 2;
                i = 0;
                while (i < linelen) {
                 p [i] = line [i];
                 i++;
         }
         /* Adjust the header. */
         d_avail[b_slot] = d_avail[b_slot]-linelen-2;
         d_lines[b_slot]++;
}
        /*
                Special case: inserting a null line at the
                end of the file is not a significant change.
         */

        if ( (linelen != 0) | (bufnrbot() == 0) ) {
                b_cflag = YES;
        }

        /* Bump the number of the last line. */
        b_max_line++;

        /* ----- check code ----- */
        check_block("bufins");
}


/*
        Combine two blocks into one if possible.
        Make the new block the current block.
 */

combine(diskp1, diskp2)
int diskp1, diskp2;
{
        char            *p1,    *p2;
        int             slot1,  slot2,  slot3;
        int             len1,   len2;
        int             i;

        /* Make sure the call makes sense. */
        if ( (diskp1 == ERROR) | (diskp2 == ERROR) ) {
                return;
        }

        /* Get the two blocks. */
        swap_in(diskp1, &slot1);
        swap_in(diskp2, &slot2);

        if ( (d_next [slot1] != diskp2) |
             (d_back [slot2] != diskp1)
           ) {
                cant_happen("combine 1");
        }

        /* Do nothing if the blocks are too large. */
        len1 = BUFF_SIZE - d_avail [slot1];
        len2 = BUFF_SIZE - d_avail [slot2];

        if ( (len1 > BUFF_SIZE) | (len2 > BUFF_SIZE) ) {
                cant_happen("combine 2");
        }

        if (len1 + len2 > BUFF_SIZE) {
                return;
        }

        /* Copy buffer 2 to end of buffer 1. */
        p1 = data_addr (slot1) + len1;
        p2 = data_addr (slot2);

        i = 0;
        while (i < len2) {
                p1 [i] = p2 [i];
                i++;


        }

        /* Both blocks are now dirty. */
        is_dirty(slot1);
        is_dirty(slot2);

        /* Adjust the back pointer of the next block. */
        if (d_next [slot2] != ERROR) {
                swap_in(d_next [slot2], &slot3);
                d_back [slot3] = d_diskp [slot1];
                is_dirty(slot3);
        }

        /*
                Adjust the current block if needed.
                The value of b_start must be decremented
                by the OLD value of d_lines [slot1].
         */

        if (b_slot == slot2) {
                b_slot  = slot1;
                b_start = b_start - d_lines [slot1];
        }

       /* Adjust the header for block 1. */
       d_lines [slot1] = d_lines [slot1] + d_lines [slot2];
       d_avail [slot1] = BUFF_SIZE - len1 - len2;
       d_next  [slot1] = d_next [slot2];

       /* Adjust the pointers to the last block. */
       if (diskp2 == b_tail) {
                b_tail = diskp1;
       }

       /* Slot 2 must remain in core until this point. */
       free_block(slot2);
```

```
        /* ----- check code ----- */
        check_block("combine");
}


/* Put the block in the slot on the free list. */

free_block(slot)
int slot;
{
        /* Link the block into the free list. */
        d_next [slot] = b_free;
        b_free = d_diskp [slot];

        /* Erase the block. */
        d_lines [slot] = 0;
        d_avail [slot] = BUFF_SIZE;
        is_dirty(slot);
}


/*
        Create a new block linked after the current block.
        Return the slot number and a pointer to the new block.
*/

/* int */
new_block (slotp)
int     *slotp;


{
        int     slot1,  slot2;
        int     diskp;

        /* Get a free disk sector. */
        if (b_free != ERROR) {

                /* Take the first block on the free list. */
                diskp = b_free;

                /* Put the block in a free slot. */
                swap_in(diskp, &slot1);

                /* Adjust the head of the free list. */
                b_free = d_next [slot1];
        }
        else {
                /* Get a free slot. */
                diskp = ++b_max_diskp;
                swap_new(diskp, &slot1);
        }

        /* Link the new block after the current block. */
        d_next [slot1] = d_next  [b_slot];
        d_back [slot1] = d_diskp [b_slot];
        d_next [b_slot] = diskp;
        if (d_next [slot1] != ERROR) {
                swap_in(d_next [slot1], &slot2);
                d_back [slot2] = diskp;
                is_dirty(slot2);
        }

        /* The block is empty. */
        d_lines [slot1] = 0;
        d_avail [slot1] = BUFF_SIZE;
        is_dirty(slot1);

        /* Set the user's field. */
        *slotp  = slot1;
}


/*
        Split the current block in two pieces.
        Length1 is the number of chars before the break point.
        Length2 is the number of chars after  the break point.
        If flag == YES, make the new block the current block.
*/

split_block(length1, length2, flag)
int length1, length2, flag;
{
        char            *p, *q;
        int             slot2;
        int             i;


        /* Create a new block. */
        new_block(&slot2);

        /* Mark both blocks as dirty. */


        is_dirty(b_slot);
        is_dirty(slot2);

        /* Copy end of the old block to start of the new. */
        p = data_addr (b_slot) + length1;
        q = data_addr (slot2);

        i = 0;
        while (i < length2) {
                q [i] = p [i];
                i++;
        }

        /* Adjust the headers. */
        d_lines [slot2] = d_lines [b_slot] - (b_line-b_start);
        d_avail [slot2] = BUFF_SIZE - length2;
        d_lines [b_slot] = b_line - b_start;
        d_avail [b_slot] = BUFF_SIZE - length1;;

        /* Adjust the pointer to the last block. */
        if (d_diskp [b_slot] == b_tail) {
                b_tail = d_diskp [slot2];
        }

        if (flag == YES) {

                /* Make the new block the current block. */
                b_start = b_start + d_lines [b_slot];
                b_slot  = slot2;
        }
}
```

```
/*
        RED buffer routines -- small-C version
        Part 3 -- file routines.

        Source:  red12.sc
        Version: August 29, 1982;  February 27, 1983

        Copyright (C) 1983 by Edward K. Ream
*/


/* Return the address of the data area for the slot. */

/* char * */
data_addr(slot)
int slot;
{
        return d_data + (slot * DATA_SIZE) + HEADER_SIZE;
}


/* Open the data file. */

/* int */
data_open()
{
        int fd;

        /* Erase the data file if it exists. */
        sysunlink(DATA_FILE);

        /* Create the data file. */
        b_data_fd = syscreat(DATA_FILE);
        if (b_data_fd == ERROR) {
                disk_error("can not open swap file.");
        }

        /* Close the file, reopen it for read/write access. */
        sysclose(b_data_fd);
        b_data_fd = sysopen(DATA_FILE, 2);
        return b_data_fd;
}


/* Make the slot the MOST recently used slot. */

do_lru(slot)
int slot;
{
        int i, lru;

        /*
                Change the relative ordering of all slots
                which have changed more recently than slot.
         */
        lru = d_lru [slot];
        i = 0;
        while (i < NSLOTS) {
                if (d_lru [i] < lru) {
                        d_lru [i]++;
                }
                i++;
        }


        /* The slot is the most recently used. */
        d_lru [slot] = 0;
}


/*
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 34)

```
        Print an error message and abort.
        It would be unwise to try to recover from here
        because the state of the work file is unknown.
        However, the work file is left intact so that it
        may be examined and the original file reconstituted.
*/

disk_error(message)
char *message;
{
        error(message);

        /* Erase the buffer for fatal errors. */
        if (b_fatal == YES) {
                bufnew();
        }

        /* Jump to the error recovery point. */
        longjmp(D_ERROR, ERROR);
}


/* Indicate that a slot must be saved on the disk. */

is_dirty(slot)
int slot;
{
        d_status [slot] = DIRTY;
}


/* Put out the block-sized buffer to the disk sector. */

put_block(buffer, sector)
char *buffer;
int sector;
{
        int s;

        /* Seek to the correct sector of the data file. */
        s = sysseek(b_data_fd, sector);
        if (s == -1) {
                disk_error("seek failed in put_block.");
        }

        /* Write the block to the data file. */
        if (syswrite( b_data_fd, buffer, READ_SIZE)
            != READ_SIZE) {
                disk_error("write failed in put_block.");
        }
}


/*
        Fill in the header fields of the output buffer and
        write it to the disk.
*/

/* char * */
put_buf(n)
int n;
{
        int     *p;

        if (n == 0) {
                cant_happen("put_buf 1");
        }

        /*
                Fill in the back and next links immediately.
                This can be done because we are not waiting
                for the LRU algorithm to allocated disk blocks.
                The last block that put_buf() writes will have
                an incorrect next link.  Read_file() will make
                the correction.

                The d_avail field calculation allows for the
                line length in the line just built.
        */

         p = b_buff;
         *p++ = b_max_diskp - 1;          /* d_back field  */
         *p++ = b_max_diskp + 1;          /* d_next field  */
         *p++ = BUFF_SIZE - n;            /* d_avail field */
         *p++ = b_line - b_start;         /* d_lines field */

        /* Update block and line counts. */
        b_max_diskp++;
        b_start = b_line;

        /* Write the block. */
        put_block(b_buff, b_max_diskp - 1);
}


/*
        Write out the slot to the data file.
 */

/* int */
put_slot(slot)
int slot;
{
        int     *p;

        if (d_diskp [slot] == ERROR) {
                cant_happen("put_slot");
        }

        /* Copy header information back to the block. */
        p    = d_data + (slot * DATA_SIZE);
        *p++ = d_back  [slot];
        *p++ = d_next  [slot];
        *p++ = d_avail [slot];
        *p++ = d_lines [slot];

        /* Write the block to the disk. */
        put_block(d_data + (slot * DATA_SIZE), d_diskp [slot]);
}

/* Read a file into the buffer. */

buf_r_file(file_name)
char file_name [];
{
int     i, j;
char    *outbuf;        /* the output buffer     */
int     in;             /* input buffer index    */
int     out;            /* output buffer index   */
int     out_save;       /* line starts here      */
int     count;          /* chars in line         */
int     c;              /* current char          */
int     *ip;            /* integer pointer       */

/* Clear the swapping buffers and the files. */
bufnew();
d_status [b_slot] = FREE;

/* Open the user file for reading only. */
b_user_fd = sysopen(file_name, 0);
if (b_user_fd == ERROR) {
        disk_error("file not found");
}

/* Open the data file. */
data_open();

/* Erase the buffer on a disk error. */
b_fatal = YES;

/* The file starts with line 1. */
b_line = 1;
b_start = 1;

/* There are no blocks in the file yet. */
b_head = b_tail = ERROR;
b_max_diskp = 0;

in = DATA_SIZE;         /* Force an initial read. */
out = 2;
out_save = 0;
b_line = b_start = 1;
outbuf = b_buff + HEADER_SIZE;
count = 0;

while (1) {

        if ((out >= BUFF_SIZE) & (out_save == 0)) {

                /* The line is too long. */
                error ("line split");

                /* End the line. */
                b_putnum(outbuf, count);
                b_line++;
                count = 0;
                        /* Clear the output buffer. */
                        put_buf(out);
                        out = 2;
                        out_save = 0;
                }

                else if (out >= BUFF_SIZE) {

                        /* Write out the buffer. */
                        put_buf(out_save);

                        /* Move the remainder to the front. */
                        i = 2;
                        j = out_save + 2;

                        while (j < out) {
                                outbuf [i++] = outbuf [j++];
                        }

                        /* Reset restart point. */
                        count = out - out_save - 2;
                        out_save = 0;
                        out = count + 2;
                }

                c = read1(&in);
```

```
                if (c == CPMEOF) {

                        if (count != 0) {

                                /* Finish the last line. */
                                b_putnum(outbuf + out_save, count);
                                b_line++;
                                out_save = out;
                        }

                        if (out_save != 0) {
                                put_buf(out_save);
                        }
                        break;
                }

                else if (c == LF) {

                        /* Ignore LF's */
                        continue;
                }

                else if (c == CR) {

                        /* Finish the line. */
                        b_putnum(outbuf + out_save, count);

                        /* Set restart point. */
                        b_line++;
                        out_save = out;
                        out = out + 2;
                        count = 0;
                }

                else {

                        /* Copy normal character. */
                        outbuf [out++] = c;
                        count++;
                }
        }

        /* Close the user' file. */
        sysclose(b_user_fd);

        /* Special case:  null file */
        if (b_max_diskp == 0) {
                bufnew();
                return;
        }

        /* Rewrite the last block with correct next field. */
        ip = b_buff;
        ip++;

        *ip = ERROR;
        put_block(b_buff, b_max_diskp - 1);

        /* Set the pointers to the first and last blocks. */
        b_max_diskp--;
        b_head = 0;
        b_tail = b_max_diskp;

        /* Move to the start of the file. */
        b_max_line = b_line - 1;
        b_line = 1;
        b_start = 1;
        swap_in(b_head, &b_slot);
        b_fatal = NO;
}


/* Get one character from the input file. */

read1(in)
int *in;
{
        char    *inbuf;
        int     s;

        /* Put the input buffer in the first slot. */
        inbuf = d_data;

        if (*in == DATA_SIZE) {

                /* Read the next sector. */
                s = sysread (b_user_fd, inbuf);
                if (s == ERROR) {
                        disk_error("read failed");
                }

                /* Force a CPM end of file mark. */
                if (s < READ_SIZE) {
                        inbuf [s * CPM_SIZE] = CPMEOF;
                }

                *in = 0;
        }

        /* Return the next character of the buffer. */
        return inbuf [(*in)++];
}


/*
        Put the block from the disk into a slot in memory.
        Return the slot number and a pointer to the block.
 */

/* int */
swap_in(diskp, slotp)
int diskp;
int *slotp;
{
        int     slot, s;
        int     *p;

        if ( (diskp < 0) | (diskp > b_max_diskp) ) {
                cant_happen("swap_in 1");
        }

        /* See whether the block is already in a slot. */
        slot = 0;
        while (slot < NSLOTS) {
                if ( (d_status [slot] != FREE) &
                     (d_diskp  [slot] == diskp)
                   ) {

                        /* Reference the block. */
                        do_lru(slot);

                        /* Set the caller's field. */
                        *slotp  = slot;
                        return;
                }
                slot++;
        }

        /* Clear a slot for the block. */
        swap_new(diskp, slotp);

        /* Seek to the proper place. */
        s = sysseek(b_data_fd, diskp);
        if (s == -1) {
                disk_error("swap_in:  disk full");
        }

        /* Read the block into the slot. */
        s = sysread(b_data_fd, d_data + (*slotp * DATA_SIZE));
        if (s == ERROR) {
                disk_error("swap_in:  disk not ready");
        }

        /* Copy information to arrays for easy access. */
        p = d_data + (*slotp * DATA_SIZE);
        d_back  [*slotp] = *p++;
        d_next  [*slotp] = *p++;
        d_avail [*slotp] = *p++;
        d_lines [*slotp] = *p++;

        /* Swap_new() has already called do_lru(). */
}
/*
        Free a slot for a block located at diskp.
        Swap out the least recently used block if required.
        Return slotp.
 */
/* int */
swap_new (diskp, slotp)
int diskp;
int *slotp;

{
        int slot;

        /* Search for an available slot. */
        slot = 0;
        while (slot < NSLOTS) {
                if (d_status [slot] == FREE) {
                        break;
                }
                slot++;
        }

        /* Swap out a block if all blocks are full. */
        if (slot == NSLOTS) {
                slot = swap_out();
        }

        /* Make sure the block will be written. */
        d_status [slot] = FULL;
        d_diskp  [slot] = diskp;

        /* Reference the slot. */
        do_lru(slot);

        /* Return slotp. */
        *slotp = slot;
}
/*
        Swap out the least recently used (LRU) slot.
        Return the index of the slot that becomes free.
 */

/* int */
swap_out()
{
        int slot;
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 34)

```
        /* Open the temp file if it has not been opened. */
        if (b_data_fd == ERROR) {
                b_data_fd = data_open();
        }

        /* Find the least recently used slot. */
        slot = 0;
        while (d_lru [slot] != NSLOTS - 1) {
                slot++;
        }

        /* Do the actual swapping out if memory is dirty. */
        if (d_status [slot] == DIRTY) {
                put_slot(slot);
                return slot;
        }

        /* d_diskp is not ERROR if status is not DIRTY. */
        if (d_diskp [slot] == ERROR) {
                cant_happen("swap_out");
        }

        /* Indicate that the slot is available. */
        d_status [slot] = FREE;
        d_diskp  [slot] = ERROR;

        /* Return the slot number. */
        return slot;
}

/* Write the entire buffer to file. */

buf_w_file(file_name)
char *file_name;
{
        char *data;
        int out, slot, lines, length, next, count;
        int c;

        /* Open the user file.  Erase it if it exists. */
        b_user_fd = syscreat(file_name);
        if (b_user_fd == ERROR) {
                disk_error("create failed");
        }

        /* Copy each block of the file. */
        out = 0;
        next = b_head;
        while (next != ERROR) {

                /* Swap in the next block. */
                swap_in(next, &slot);

                /* Get data from the header of the block. */
                next  = d_next [slot];
                lines = d_lines [slot];
                data  = data_addr (slot);

                /* Copy each line of the block. */
                count = 0;
                while (lines--) {

                        /* Get length of the line. */
                        length = b_getnum(data + count);

                        /* Skip over length field. */
                        count = count + 2;

                        /* Copy each char of the line. */
                        while (length--) {
                                c = data [count++];
                                write1(c, &out);
                        }

                        /* Add CR and LF at end. */
                        write1(CR, &out);
                        write1(LF, &out);
                }
        }

        /* Force an end of file mark. */
        write1(CPMEOF, &out);

        /* Flush the buffer and close the file. */
        write_flush(out);
        sysclose(b_user_fd);

        /* Kludge:  go to line 1 for a reference point. */
        swap_in(b_head, &b_slot);
        b_line = b_start = 1;
}


/*
        Write one character to the user's file.
        i is the current position in the file buffer.
*/

/* int */
write1(c, i)
char c;
int *i;
{
        b_buff [(*i)++] = c;
        if (*i == CPM_SIZE) {

                if (syswrite(b_user_fd, b_buff, 1) != 1) {
                        disk_error("write failed");
                }
                *i = 0;
        }
}

/*
        Flush b_buff to the user's file.
*/

write_flush(i)
int i;
{
        if (i == 0) {
                return;
        }
        if (syswrite(b_user_fd, b_buff, 1) != 1) {
                disk_error("flush failed");
        }
}
```

```
/*
        RED buffer routines -- small-C version
        Part 4 -- debugging routines.

        Source:  red13.sc
        Version: August 29, 1982;  February 26, 1983

        Copyright (C) 1983 by Edward K. Ream
*/

/*
        You may omit this file if you need to save space.
        Just delete the calls to check_block() in the
        file red12.sc.  Then delete the #include red13.sc
        statement in file red.sc.
*/

/*
        The observer effect:  None of the routines of this file
        should ever call swap_in() because swap_in() causes
        all kinds changes to the data structures which these
        routines are trying to print out.
*/

/*
        Dump global varialbes, all resident slots and
        the current block.
*/

bufdump()
{
        dump_vars();
        dump_memory();
        dump_block();

}
cant_happen(message)
char *message;
{
        pp(message);
        ppl(":  can't happen");

        bufdump();
        exit();
}
/* Check the current block for consistency. */

check_block(message)
char *message;
{
        int count, i, total;
        char *cp;

        if ( (b_slot == ERROR) |
             (b_line < 0)      |
             (b_line > b_max_line + 1)
           ) {

                ppl("check block 1");
                cant_happen(message);
        }

        if ( (d_lines [b_slot] < 0)          |
             (d_lines [b_slot] >= BUFF_SIZE) |
             (d_avail [b_slot] < 0)          |
             (d_avail [b_slot] > BUFF_SIZE)
           ) {

                ppl("check block 2");
                cant_happen(message);
        }

        /* Make sure there are at least enough lines. */
        cp    = data_addr (b_slot);
        total = 0;
```

```
        i = 0;
        while (i++ < d_lines [b_slot]) {
                count = b_getnum(cp);
                total = total + count + 2;
                cp    = cp + count + 2;

                if ( (count < 0)          |
                     (count > BUFF_SIZE)  |
                     (total > BUFF_SIZE)
                   ) {

                        ppl("check block 3");
                        cant_happen(message);
                }
        }
}
/* Dump the current block. */

dump_block()
{
        int lines, c, count, i, j, total;
        char *cp;

        ppl("");
        ppl("Dump of current block:");
        ppl("");

        lines = d_lines [b_slot];
        cp    = data_addr (b_slot);
        total = 0;

        i = 0;
        while ( (i < lines) & (total <= BUFF_SIZE) ) {
                count = b_getnum(cp);
                total = total + 2;
                cp    = cp + 2;

                pp("line ");
                ppdec(i + 1, 3);
                pp(", count ");
                ppdec(count, 3);
                pp(", total ");
                ppdec(total, 3);
                pp(":  ");

                if ( (count < 0) | (total < 0) ) {
                        return;
                }

                /* Print each line of the block. */

                j = 0;
                if (count > 50) {
                        ppl("");
                }
                while ( (j < count) & (j < 80) ) {

                        c = cp [j] & 127;

                        if (c == TAB) {
                                ppc(' ');
                        }
                        else if (c < 32) {
                                ppc('^');
                                ppc(c + 64);
                        }
                        else {
                                ppc(c);
                        }

                        total++;
                        if (total >= BUFF_SIZE) {
                                break;
                        }
                        j++;
                }
                ppl("");
                cp = cp + count;

                i++;
        }
}
/* Dump all the resident slots. */

dump_memory()
{
        int             i;

        ppl("Dump of slots:");
        ppl("");

        i = 0;
        while (i < NSLOTS) {

                pp("slot ");
                ppdec(i, 2);
                pp(", diskp ");
                ppdec(d_diskp [i], 6);
                pp(", back ");
                ppdec(d_back [i], 6);
                pp(", next ");
                ppdec(d_next [i], 6);
                pp(", avail ");
                ppdec(d_avail [i], 6);
                pp(", lines ");
                ppdec(d_lines [i], 6);
                pp(", lru ");
                ppdec(d_lru [i], 3);
                pp(", status ");
                ppdec(d_status [i], 3);
                ppl("");
                i++;
        }
}

/* Dump all global variables. */

dump_vars()
{
        pp("slot ");
        ppdec(b_slot, 3);
        pp(", maxdiskp ");
        ppdec(b_max_diskp, 3);
        pp(", line ");
        ppdec(b_line, 5);
        pp(", maxline ");
        ppdec(b_max_line, 5);
        pp(", start ");
        ppdec(b_start, 3);
        pp(", head ");
        ppdec(b_head, 3);
        pp(", tail ");
        ppdec(b_tail, 3);
        pp(", free ");
        ppdec(b_free, 3);
        ppl("");
        pp("back ");
        ppdec(d_back [b_slot], 3);
        pp(", next ");
        ppdec(d_next [b_slot], 3);
        pp(", avail ");
        ppdec(d_avail [b_slot], 3);
        pp(", lines ");
        ppdec(d_lines [b_slot], 3);
        pp(", lru ");
        ppdec(d_lru [b_slot], 3);
        pp(", status ");
        ppdec(d_status [b_slot], 3);
        pp(", diskp ");
        ppdec(d_diskp [b_slot], 3);
        ppl("");
}
/* Output a string to the dump device (console). */

pp(str)
char * str;
{
        while(*str) {
                syscout(*str++);
        }
}
/* Output a string to the dump device, followed by a CR/LF. */

ppl(str)
char * str;
{
        pp(str);
        syscout(13);
        syscout(10);
}
/* Print a decimal number in a field of width w. */

ppdec(number, w)
int number;
int w;
{
        int count, k, zs;
        char c, buffer[10];

        count = 0;
        zs = 0;
        k = 10000;
        if (number < 0) {
                number = -number;
                buffer [count++] = '-';
        }
        while (k >= 1) {
                c = number/k + '0';
                if ( (c != '0') | (k == 1) | (zs == 1)) {
                        zs = 1;
                        buffer [count++] = c;
                }
                number = number%k;
                k = k/10;
        }
        buffer [count] = 0;
        while (count++ < w) {
                syscout(' ');
        }
        pp(buffer);
}


/* Print one character to the dump device (console). */

ppc(c)
char c;
{
        syscout(c);
}
```

End Listings

Anatomy of a Digital Vector and Curve Generator

Have you ever wondered how your computer draws a line on your TV set? All you tell the computer is endpoints of the line, and the CPU does the rest. For example,

```
PLOT 0,0 : DRAWTO 310,159
```

will draw a line from the upper left corner of the screen to the lower right corner. To fill in all the points on the line in between the endpoints is exactly what the DDA (Digital Differential Analyzer) algorithm does.

In this article, I will try to explain how the Unit Increment DDA vector generator works – also how Marvin Minsky's circle generator works. I have included BASIC programs for both the vector and circle generators. Both of these curve (a straight line is a special case) generators are incremental, meaning that point N+1 is generated from point N.

The DDA Vector Generator

The DDA solves the differential equation for a straight line between given endpoints. The equation used by the DDA is

$$dy/dx = (y_1 - y_0)/(x_1 - x_0)$$

This says that the slope of the line is equal to the difference of the y coordinates of the endpoints divided by the difference of the x coordinates. Note: dy means a very small change in y and dx means a very small change in x.

Given the endpoints (x_0,y_0) and (x_1,y_1), $DX=x_1-x_0$ and $DY=y_1-y_0$. If the slope is less than 45 degrees, as in Figure 1 (at right), then ABS (DY/DX) is less than 1. Let the estimate of the line length be N=abs(DX). Under the above constraints, the routine (in pseudocode) for the incremental solution is:

```
for I=0 to N-1 do
    begin
      X(I+1)=1*sgn(DX)+X(I);
      Y(I+1)=abs(DY/DX)*sgn(DY)+Y(I);
    end;
for I=0 to N do
    begin
      plot(X(I),Y(I));
    end;
```

When N, the line length estimate, equals abs(DX), and the x increment is 1, the "for" statement will trace out all the points between the endpoints of the line.

If the angle is greater than 45 degrees, we flip the places of X and Y in the above routine. The BASIC program shown in Listing 1 (page 67) handles all cases of endpoint location.

It's important to note that abs(DY/DX) is always less than or equal to 1. If this constraint is violated, then the role of the x and y coordinates must be reversed, and abs(DY) is used as the line length estimate.

Minsky's Circle Generator

Figure 2 (page 66) shows a circle with a set of x,y axes. We will consider the points (x,0) and (0,y), since these two points form the basis of vectors that span all of two-dimensional space. Take the point (x,0) and consider what happens when that point is rotated by a very small amount, α. This will twist point (x,0) into $(x,\alpha*x)$. Now point (0,y) will be twisted (at the same time) into the point $(\alpha*y,y)$. As you can see, the amount of twist is proportional to the point's distance from the origin. This is a way of describing circular motion.

Now we form the following matrix equation:

$$P(n+1) = p(n)*T(\alpha),$$

and

$$P(i) = (x_i,y_i),$$

a point $T(\alpha)=\begin{pmatrix} 1 & \alpha \\ -\alpha & 1 \end{pmatrix}$

so we now have:

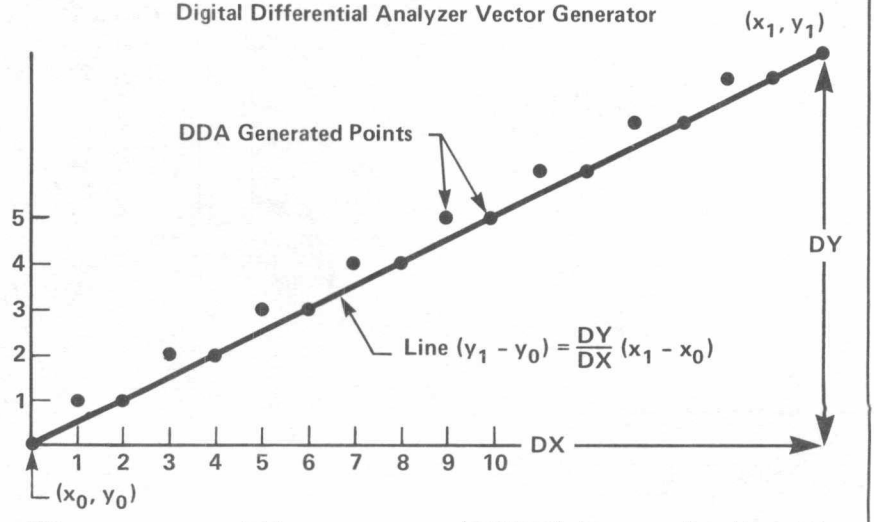
$$(x(n+1),y(n+1)) = (y(n),y(n))\begin{pmatrix} 1 & \alpha \\ -\alpha & 1 \end{pmatrix}$$

This is close to what we want, but it has one problem: The determinant of matrix T is not equal to 1. If $\det(T)>1$, then the matrix will generate a spiral of increasing radius. If $\det(T)<1$, then the curve generated is a decreasing radial spiral. So only when $\det(T)=1$ does the curve close on itself. Let's try changing T so $\det(T)=1$.

let $T(\alpha)=\begin{pmatrix} 1 & \alpha \\ -\alpha & 1-\alpha^2 \end{pmatrix}$

This equation has all the properties we are looking for; α is the angular velocity of the transition of point P(n) into point P(n+1). α is a small number less than or equal to 1. If $\alpha=1$ then a closed curve with six equally-spaced points is generated. The smaller α is, the closer the points of the curve will be, and the rounder the cir-

Figure 1.
Digital Differential Analyzer Vector Generator



$DX = x_1 - x_0$; $DY = y_1 - y_0$; $|DY/DX| < 1$; $|DX| > |DY|$
Therefore, the x increment equals 1 times SGN(DX), and the y increment equals |DY/DX| times SGN(DY). Let N equal |DX|. *Note:* N equals the largest delta.

by Richard Shiffman

Richard Shiffman, USC Information Science Institute, 4676 Admiralty Way, Marina Del Rey, California 90291.

cle will be.

Now that we have the matrix form of the circle generator, let's get it into iterative form for use by a program. Converting the form matrix equation to simultaneous linear equations gives the following.

(1) $x(n+1)=x(n)-\alpha*y(n)$
(2) $y(n+1)=\alpha*x(n)+(1-\alpha^2)*y(n)$

These two equations let you generate the point $(x(n+1),y(n+1))$ from the point $(x(n),y(n))$, but this still involves a good deal of arithmetic. You have to square α and subtract it from 1, then multiply it by $y(n)$. To speed up the circle generator, we will solve equation 1 for $x(n)$ and substitute it into equation 2. Performing the arithmetic yields:

(1) $x(n+1)=x(n)-\alpha*y(n)$
(2) $y(n+1)=\alpha*x(n+1)+y(n)$

This is what we were after all the time. Writing the above in BASIC, we get the following program:

```
10  FOR I=0 TO N
20        X=X-A*Y
30        Y=A*X+Y
40        PLOT X,Y
50  NEXT I
```

which is the heart of the program in Listing 2 (page 67).

A closing comment about the circle generator. If α is of the form $1/2^n$, the multiplication can be replaced by a shift operation. This makes the routine well-suited for a hardware implementation.

References

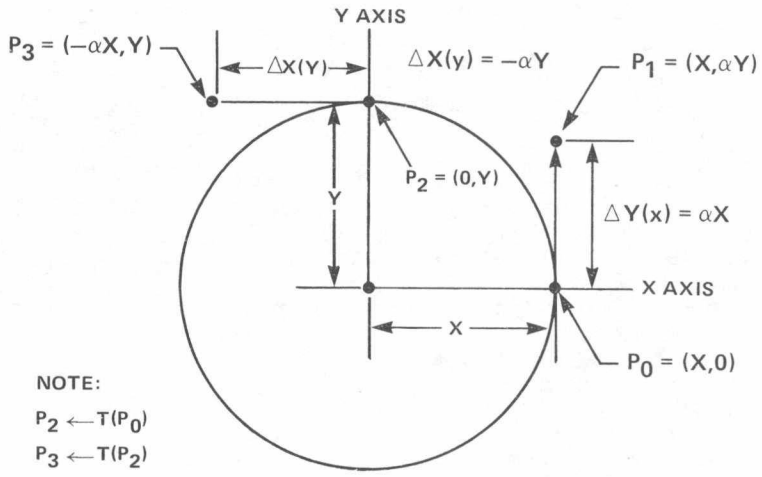
(1) *Principles of Interactive Computer Graphics,* Newman & Sproull, McGraw-Hill Book Company.

(2) Hakmem, Item 149, Marvin Minsky, M.I.T. Artificial Intelligence Lab., AIM-239, Feb-1972.

DDJ

(Listing begins at right)

Figure 2.
Incremental Circle Generator



Let $\overline{P}_{n+1} = \overline{P}_n \overline{\overline{T}}$, $\overline{\overline{T}} = \begin{pmatrix} 1 & -\alpha \\ \alpha & \psi \end{pmatrix}$, $\overline{P}_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}$

Note: For $\overline{\overline{T}}$ to generate a closed curve, DET(T) must equal 1, therefore ψ must equal $(1-\alpha^2)$

So: $\begin{pmatrix} x_n \\ y_n \end{pmatrix}\begin{pmatrix} 1 & -\alpha \\ \alpha & (1-\alpha^2) \end{pmatrix} = \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix}$ or: (1) $x_{n+1} = x_n - \alpha y_n$ (2) $y_{n+1} = \alpha x_n + (1-\alpha^2) y_n$

Solve (1) for x_n: $x_n = x_{n+1} + \alpha y$

Substitute (1) into (2): $y_{n+1} = \alpha(x_{n+1} + \alpha y_n) + (1-\alpha^2) y_n$

So: $y_{n+1} = \alpha x_{n+1} + \alpha^2 y_n + y_n - \alpha^2 y_n = \alpha x_{n+1} + y_n$

(1) $x_{n+1} = x_n - \alpha y_n$
(2) $y_{n+1} = \alpha x_{n+1} + y_n$

Leads to the following BASIC program:

```
10  FOR   I = 0 TO N
20        X = X - A * Y
30        Y = A * X + Y
40        PLOT X,Y
50  NEXT  I
```

Unit Increment DDA

Listing One

(Text begins on page 65)

```
10 REM UNIT INCREMENT DDA
15 REM
20 REM Setup display for high res.
30 REM 320 X 160 pixels,
40 REM colors are black and white
50 GRAPHICS 8
60 SETCOLOR 1,0,14:SETCOLOR 2,0,0
70 COLOR 1
80 REM
90 REM Get the two end points for
100 REM the vector to draw.
110 ? "X0,Y0,X1,Y1"
120 INPUT X0,Y0,X1,Y1
130 REM
140 REM Plot the first endpoint
150 PLOT X0,Y0
160 REM Calculate the deltas
170 DX=X1-X0
180 DY=Y1-Y0
190 SX=SGN(DX):REM Direction of X incr
200 SY=SGN(DY):REM Direction of Y incr
210 IF ABS(DX) < ABS(DY) THEN GOTO 350
220 REM ABS(DX) > ABS(DY)
230 REM Increment X axis by 1 unit
240 REM Increment Y axis by fraction
250 DYDX=ABS(DY/DX)
260 N=ABS(DX):REM N=Length estimate
270 FOR CNT=1 TO N
280 X0=X0+1*SX
290 Y0=Y0+DYDX*SY
300 PLOT INT(X0),INT(Y0+0.5)
310 NEXT CNT:GOTO 110
320 REM ABS(DX) <= ABS(DY)
330 REM Increment Y axis by 1 unit
340 REM Increment X axis by fraction
350 N=ABS(DY):REM Line length est.
360 DXDY=ABS(DX/DY)
370 FOR CNT=1 TO N
380 Y0=Y0+1*SY
390 X0=X0+DXDY*SX
400 PLOT INT(X0+0.5),INT(Y0)
410 NEXT CNT
420 GOTO 110
```

Incremental Circle Generator

Listing Two

```
10 REM INCREMENTAL CIRCLE GENERATOR
15 REM
20 REM This routine will generate an
30 REM almost circle for alpha <= 1.
40 REM Alpha is the angular velocity
50 REM of the points making up the
60 REM circle. The smaller alpha is
70 REM the rounder the circle.
80 REM
90 REM Set display to 320x160 pixels
```

Incremental Circle Generator

Listing Two

(Listing continued, text begins on page 65)

```
100 REM Set color to black and white
110 GRAPHICS 8
120 SETCOLOR 2,0,0:SETCOLOR 1,0,14
130 COLOR 1
140 REM
150 REM CX,CY is the center of the screen
160 CX=160:CY=80
170 REM
180 REM The smaller alpha is, the more
190 REM more points that are needed
200 REM to close the circle.
210 ? "INPUT RADIUS, ALPHA, # OF POINT"
220 INPUT R,A,N
230 Y=R:X=0
240 REM
250 REM The following is the main loop
260 REM of the circle generator.
270 FOR I=0 TO N
280 X=X-A*Y
290 Y=A*X+Y
300 PLOT X+CX,Y+CY
310 NEXT I
320 STOP
330 GOTO 210
```

End Listing Two

Fast Matrix Operations
in Forth, Part II

by Steven A. Ruzinsky

In Part I of this series (*DDJ* No. 80, June 1983), a matrix-defining word and some fundamental accessories were presented. In this installment, programs for performing matrix operations and some more-advanced accessories are presented. In order to use the programs shown in screens 210 to 227 (pages 73-79), the programs given in Part I, i.e., screens 207 to 209, must be preloaded.

Several points from Part I deserve reiteration. These programs are in Polyforth for the IBM PC from Forth, Inc. An 8087 numeric coprocessor must be installed in the IBM PC. Since the data contained in the matrices reside outside the Polyforth dictionary (beyond the first 64 KB segment of memory), more than 64 KB of RAM is needed. Application programs that will be presented in Part III will require 320 KB.

Regarding programming style, the programs have been written mostly in Polyforth assembly language rather than Polyforth proper in order to maximize speed. Typically, numeric programs such as these run five times faster in assembly than in Polyforth. Benchmark times are shown in Figure 1 (at right). Two timings are given for an operation, one for 10th-order square matrices and another for 50th-order square matrices. It is interesting to compare these timings with those resulting from similar programs involving different language and hardware combinations. For example, the matrix inversion program, INV, is 110 times faster than an identical algorithm implemented in the standard BASIC for the IBM PC.

Although there was little compromise in trading off program compactness for program speed, it should be noted that the entire set of programs, screens 207 to 227, occupies only 5186 bytes of RAM. Considering this compactness and the widespread applicability of matrix operations, I believe that it is not unreasonable to suggest that a refinement of these operations, perhaps more in the style of APL, should be made a standardized superset of the Forth language. Unfortunately, the Forth community, in general, seems to have only a casual interest in floating-point applications. A "fixed-point philosophy" has been widely promoted and justified in terms of a speed superiority of integer arithmetic over floating-point arithmetic. With an 8087-equipped computer, this rationale simply does not apply. Most of the 64-bit floating-point operations such as F* and FSWAP are actually faster than their 16-bit integer counterparts. If numeric coprocessors similar to the 8087 become common in future microcomputers, a fixed-point philosophy would be totally unjustified for Forth.

Program descriptions follow. In the usage examples, it is assumed that matrices X, Y, and Z have been defined and are of suitable dimensions. Algebraic equivalents to the usage examples are given in terms of X_1, X_2, Y_1, Y_2, Z_1, and Z_2, where the subscripts refer to the first and second of two interlaced submatrices comprising each matrix.

AA, AA!, AA@ – These are, respectively, surrogate matrix, surrogate matrix store, and surrogate matrix fetch. After initialization by either of the two phrases,

```
' X adr!
' X dim@
```

AA becomes the functional equivalent of X, AA! becomes the functional equivalent of X !1, and AA@ becomes the functional equivalent of X @1.

A!, R!, C! – Respectively, matrix store, row store, and column store. Following execution, input into the matrix is prompted. Usage examples:

```
' X A!
' X 3 R!
' X 5 C!
```

A., R., C. – Display, respectively, a matrix, a matrix row, and a matrix column. Usage examples:

```
' X A.
' X 2 R.
' X 7 C.
```

A1!, R1!, C1!, A1., R1., C1. – Similar to the preceding except that they apply only to the first of two interlaced matrices.

a-! – Stores the number on the numeric stack into every element of a matrix. Usage example:

```
3.14 ' X a-!
```

excA1, excR1, excC1 – Exchange, respectively, matrices, matrix rows, and

Operation	Comment	Time, Seconds 10x10 Matrices	50x50 Matrices
a-!		0.010	0.072
excA1	exchange	0.022	0.218
->	transfer	0.026	0.121
exc12		0.015	0.197
TRNSP	transposition	0.017	0.113
a*	scalar mult.	0.014	0.192
A+	addition	0.020	0.191
A*	multiplication	0.099	9.822
T*		0.100	9.825
INV	inversion	0.217	22.179
DET	determinant	0.074	4.969
sortA1	sort	0.106	5.824
sortA	double sort	0.139	7.908
polyA1		0.013	0.151
rndA	random no. fill	0.032	0.612

Figure 1.
Polyforth Matrix Operation Benchmarks

Steven A. Ruzinsky, 2110 S. Austin Blvd., Cicero, IL 60650.

matrix columns. Usage examples:

```
' X ' Y excA1
' X ' Y 5 10 excR1
' X ' Y 4 4 excC1
```

—> – Matrix transfer. Usage example:

```
' X ' Y 1 2 —>
```

In algebraic notation, $Y_2 = X_1$.

exc12 – Exchanges interlaced matrices. Usage example:

```
' X exc12
```

TRNSP – Matrix transposition. Usage example:

```
' X ' Y TRNSP
```

In algebraic notation, $Y_1 = X_1^T$.

a* – Multiplies every element of a matrix by the number on the numeric stack. Usage example:

```
12.34 ' X a*
```

A+ – Adds two matrices. Usage example:

```
' X ' Y A+
```

In algebraic notation, $Y_1 = Y_1 + X_1$.

A* – Multiplies two matrices. Usage example:

```
' X ' Y ' Z A*
```

In algebraic notation, $Z_1 = X_1 \; Y_1$.

T* – Transposes a matrix and then multiplies it by a second matrix. Usage example:

```
' X ' Y ' Z T*
```

In algebraic notation, $Z_1 = X_1^T \; Y_1$.

INV – Inverts a matrix. Usage example:

```
' X INV
```

In algebraic notation, $X_1 = X_1^{-1}$

DET – Finds the determinant of a matrix and puts the result on the numeric stack. Usage example:

```
' X DET
```

sortA, sortA1 – Sorts the elements of a matrix in ascending order. Usage example:

```
' X sortA
```

The sorting is performed as indicated by the following arrows:

$$X = \left[\begin{array}{c} \downarrow \\ \downarrow \\ \downarrow \end{array} \right]$$

sortA1 sorts X_1 only, whereas sortA sorts both X_1 and X_2 with respect to the values in X_1.

polyR1 – Enters the elements of a polynomial into a matrix row. Usage example:

```
2.0 ' X 5 polyR1
```

Here, the following values will be entered in row 5 of the matrix as indicated:

$$X = \left[\begin{array}{cccccc} 1 & 2 & 4 & 8 & 16 & \ldots \\ & & & & & \end{array} \right]$$

polyV1 – Enters the elements of a polynomial into a column vector, i.e., an m-by-1 matrix. Usage example:

```
3.0 ' X polyV1
```

Here:

$$X = \left[\begin{array}{c} 1 \\ 3 \\ 9 \\ 27 \\ . \end{array} \right]$$

polyA1 – Starting with a matrix, e.g., X, such that

$$X = \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \\ \end{array} \right]$$

The phrase

```
' X polyA1
```

results in

$$X = \left[\begin{array}{ccccc} 1 & x_1 & x_1^2 & x_1^3 & \ldots \\ 1 & x_2 & x_2^2 & x_2^3 & \ldots \\ 1 & x_3 & x_3^2 & x_3^3 & \ldots \\ 1 & x_4 & x_4^2 & x_4^3 & \ldots \\ . & . & . & . & \ldots \\ & & & & \end{array} \right]$$

V* – Multiplies two column vectors, i.e., m x 1 matrices. Usage example:

```
' X ' Y V*
```

The result is placed on the numeric stack.

rndA – Fills a matrix with uniformly distributed pseudo-random numbers. Usage example:

```
' X rndA
```

In addition to the above matrix operations, a few fundamental scalar operations missing from my version of Polyforth have been included. *[We understand that the most recent release of Polyforth now includes a complete trig package. – Ed.]* They are:

SIN – Sine function (angle must be in radians)

COS – Cosine function (angle must be in radians)

RND – Places a uniformly distributed pseudo-random number between 0 and 1 on numeric stack

RANDOMIZE – Reseeds random number generator

This concludes Part II of a three-part series. In Part III, the operations presented in the first two parts will be used to build some application programs. ■■J

(Listing begins at right)

Fast Matrix Operations

(Text begins on page 70)

```
210 LIST

  0 ( One Dimensional Arrays and Random Number Generator )
  1 SVARIABLE aa SVARIABLE cc SVARIABLE mm LVARIABLE xx
  2 24298.0 aa S! 99991.0 cc S! 199017.0 mm S! 12345.0 xx L!
  3 ASSEMBLER CREATE
  4 (RND) FLD1 R32 aa FLD R64 xx FMUL R32 cc FADD R32 mm FDIV
  5 REV FPREM 1 N) FSTP 0 N) FLD R32 mm FMUL R64 xx FSTP RET
  6 CODE RND (RND) CALL NEXT
  7 : RANDOMIZE  ." Seed = " IN# xx L! ;
  8 : ARRAY CREATE 1+ 2* ALLOT ;CODE 0 POP 0 SHL W 0 ADD
  9          0 INC 0 INC 0 PUSH NEXT
 10 : T. COUNTER SWAP - >N 1000.0 F/ 3 F. ." sec.";
 11 : START COUNTER ;
 12 CODE (rndA) m LDA n MUL 0 1 MOV adr LDA 65520 # W MOV
 13 BEGIN (RND) CALL 0 DS LSG R64 W ) FSTP
 14 2 2 SUB 2 DS LSG 0 INC LOOP NEXT
 15 : rndA dim@ n! m! (rndA) ;


211 LIST

  0 ( SIN and COS  )
  1 LVARIABLE S1 .9999999999999999 S1 L!
  2 CODE COS FLD1 FLDPI FSCALE 1 N) FSTP 1 FXCH FPREM 1 N) FSTP
  3 FABS FLD1 FCHS FLDPI 2 N) FSUB -POP FSCALE 1 N) FSTP 1 FXCH
  4 FABS 1 N) FSUB REV R64 S1 FMUL FPTAN 1 N) FLD 0 N) FMUL NO
  5 1 FXCH 0 N) FMUL NO 1 N) FADD FSQRT 1 N) FDIV NEXT
  6 CODE SIN FLD1 FCHS FLDPI FSCALE 1 N) FSTP 1 N) FSUB
  7 FLD1 FLDPI FSCALE 1 N) FSTP 1 FXCH FPREM 1 N) FSTP
  8 FABS FLD1 FCHS FLDPI 2 N) FSUB -POP FSCALE 1 N) FSTP 1 FXCH
  9 FABS 1 N) FSUB REV R64 S1 FMUL FPTAN 1 N) FLD 0 N) FMUL NO
 10 1 FXCH 0 N) FMUL NO 1 N) FADD FSQRT 1 N) FDIV NEXT
 11
 12
 13
 14
 15


212 LIST

  0 ( Surrogate Matrix Fetch and Store, Initialized by dim@ )
  1 CODE AA! 0 POP m MUL 2 POP 0 2 ADD adr LDA 0 2 ADD 1 DS SSG
  2          2 DS LSG R64 65520 FSTP 1 DS LSG NEXT
  3 CODE AA@ 0 POP m MUL 2 POP 0 2 ADD adr LDA 0 2 ADD 1 DS SSG
  4          2 DS LSG R64 65520 FLD 1 DS LSG NEXT
  5 CODE AA 0 POP m MUL 2 POP 0 2 ADD adr LDA 0 2 ADD 2 PUSH NEXT
  6
  7
  8
  9
```

(Continued on next page)

Fast Matrix Operations

(Listing continued, text begins on page 70)

```
 10
 11
 12
 13
 14
 15
```

```
213 LIST

  0 ( Interlaced Matrix and Row and Column Input and Output )
  1 : C! j! a! a@ dim@ DROP m! m@ 0 DO CR ." a( " I . ." , "
  2         j@ . ." ) = " IN# ." i " IN# I j@ AA !! LOOP ;
  3 : C. j! a! a@ dim@ DROP m! m@ 0 DO CR ." a( " I . ." , "
  4    j@ . ." ) = " I j@ AA @@ FSWAP N. ." i" N. LOOP ;
  5 : R! i! a! a@ dim@ n! DROP n@ 0 DO CR ." a( " i@ . ." , "
  6         I . ." ) = " IN# ." i " IN# i@ I AA !! LOOP ;
  7 : R. i! a! a@ dim@ n! DROP n@ 0 DO CR ." a( " i@ . ." , "
  8    I . ." ) = " i@ I AA @@ FSWAP N. ." i" N. LOOP ;
  9 : A. a! a@ dim@ DROP m! m@ 0 DO CR ." Row " I . a@ I R. LOOP ;
 10 : A! a! a@ dim@ DROP m! m@ 0 DO CR ." Row " I . a@ I R! LOOP ;
 11
 12
 13
 14
 15
```

```
214 LIST

  0 ( Matrix and Matrix Row and Column Input and Output )
  1 : C1! j! a! a@ dim@ DROP m! m@ 0 DO CR ." a( " I . ." , "
  2         j@ . ." ) = " IN# I j@ AA! LOOP ;
  3 : C1. j! a! a@ dim@ DROP m! m@ 0 DO CR ." a( " I . ." , "
  4         j@ . ." ) = " I j@ AA@ N. LOOP ;
  5 : R1! i! a! a@ dim@ n! DROP n@ 0 DO CR ." a( " i@ . ." , "
  6         I . ." ) = " IN# i@ I AA! LOOP ;
  7 : R1. i! a! a@ dim@ n! DROP n@ 0 DO CR ." a( " i@ . ." , "
  8    I . ." ) = " i@ I AA@ N. LOOP ;
  9 : A1. a! a@ dim@ DROP m! m@ 0 DO CR ." Row " I . a@ I R1.
 10         LOOP ;
 11 : A1! a! a@ dim@ DROP m! m@ 0 DO CR ." Row " I . a@ I R1!
 12         LOOP ;
 13
 14
 15
```

```
215 LIST

 0 ( Matrix-Scalar Multiplication; Matrix Fill, Clear; Subrs. )
 1 CODE (a*) m LDA n MUL 0 1 MOV adr LDA 65520 # W MOV
 2 BEGIN 0 N) FLD 0 DS LSG R64 W ) FMUL R64 W ) FSTP 0 INC LOOP
 3 0 N) FSTP 2 2 SUB 2 DS LSG NEXT
 4 : a* dim@ n! m! (a*) ;
 5 CODE (a-!) m LDA n MUL 0 1 MOV adr LDA 65520 # W MOV
 6 BEGIN 0 DS LSG R64 W ) FST 0 INC LOOP 0 N) FSTP 2 2 SUB
 7 2 DS LSG NEXT
 8 : a-! dim@ n! m! (a-!) ; : clr 0.0 a-! ;
 9 : <square> dim@ n! n@ = NOT ABORT" Not Square " ;
10 CODE (I2!) I PUSH R PUSH 65528 # I MOV n LDA adr 2 MOV
11 2 W MOV 0 1 MOV FLD1 BEGIN 1 R MOV
12 0 1 MOV FLDZ BEGIN 2 DS LSG R64 I ) FST 2 INC LOOP
13 0 N) FSTP W DS LSG R64 I ) FST 0 W ADD W INC R 1 MOV LOOP
14 0 N) FSTP 2 2 SUB 2 DS LSG R POP I POP NEXT
15 : I2! <square> (I2!) ;
```

```
216 LIST

 0 ( Exchange Rows,  e.g. ' X ' Y 5 3 excR1 )
 1 ~ CODE excR1 2 POP 0 POP 1 POP W POP I PUSH 1 I MOV W ) 0 ADD
 2 I ) 2 ADD 4 W) 1 MOV 2 W) W MOV 65520 # I MOV BEGIN 0 DS LSG
 3 R64 I ) FLD 2 DS LSG R64 I ) FLD 0 DS LSG R64 I ) FSTP 2 DS LSG
 4 R64 I ) FSTP W 0 ADD W 2 ADD LOOP 2 2 SUB 2 DS LSG I POP NEXT
 5 ( Exchange Columns, e.g. ' X ' Y 4 7 excC1 )
 6 ~ CODE excC1 U 0 MOV U POP R 1 MOV R POP I 2 MOV I POP W POP
 7 0 PUSH 1 PUSH 2 PUSH 2 W) 1 MOV U 0 MOV 2 W) MUL W ) 0 ADD
 8 0 W MOV R 0 MOV 2 I) MUL I ) 0 ADD 65504 # I MOV BEGIN 0 INC
 9 W INC 0 DS LSG R64 I ) FLD W DS LSG R64 I ) FLD 0 DS LSG
10 R64 I ) FSTP W DS LSG R64 I ) FSTP LOOP 2 2 SUB 2 DS LSG
11 I POP R POP U POP NEXT
12 : <dim> b! b@ dim@ n! m! adr@ c! a! a@ dim@ n@ = SWAP m@ = AND
13 NOT ABORT" Inequal Dimensions " ;
14 ( Exchange Matrices, e.g. ' X ' Y excA1 )
15 ~ : excA1 <dim> n@ 0 DO a@ b@ I I excR1 LOOP ;
```

```
217 LIST

 0 ( Matrix Addition 10th-.019 sec. 50th-.165 sec.  )
 1 CODE (A+) m LDA n MUL 0 1 MOV adr LDA c 2 MOV
 2 65520 # W MOV BEGIN 0 DS LSG R64 W ) FLD 2 DS LSG R64
 3 W ) FADD R64 W ) FSTP 0 INC 2 INC LOOP 2 2 SUB 2 DS LSG NEXT
 4 : A+ <dim> (A+) ;
 5 ( Matrix Transfer, e.g. ' X ' Y 1 2 -> )
 6 CODE (->) I PUSH m LDA n MUL 0 1 MOV adr LDA c 2 MOV
 7 j I MOV j W MOV BEGIN 0 DS LSG R64 I ) FLD 2 DS LSG
```

(Continued on next page)

Fast Matrix Operations

(Listing continued, text begins on page 70)

```
 8 R64 W ) FSTP 0 INC 2 INC LOOP 2 2 SUB 2 DS LSG I POP NEXT
 9 : -> 2SWAP <dim> 1 = IF 65520 ELSE 65528 THEN j! 1 = IF 65520
10 ELSE 65528 THEN i! (->) ;
11 ( Exchange of Interlaced Matrices, e.g. ' X exc12 )
12 ~ CODE (exc12) I PUSH m LDA n MUL 0 1 MOV adr LDA 65520 # I MOV
13 65528 # W MOV DS PUSHS BEGIN 0 DS LSG R64 I ) FLD R64 W ) FLD
14 R64 I ) FSTP R64 W ) FSTP 0 INC LOOP DS POPS I POP NEXT
15 ~ : exc12 dim@ n! m! (exc12) ;
```

```
218 LIST

 0 ( Matrix Transposition, 10th-.017 sec.  50th-.119 sec. )
 1 CODE (TRNSP) I PUSH R PUSH U PUSH 65520 # I MOV n R MOV m LDA
 2 0 U MOV R MUL 0 DEC adr 2 MOV 'c W MOV R 1 MOV
 3 BEGIN 1 ES LSG U 1 MOV BEGIN 2 DS LSG R64 I ) FLD W DS LSG
 4 R64 I ) FSTP 2 INC R W ADD LOOP 0 W SUB 1 ES SSG LOOP 2 2 SUB
 5 2 ES LSG 2 DS LSG U POP R POP I POP NEXT
 6 ~ : TRNSP b! a! b@ dim@ m! n! adr@ c! a@ dim@ n@ = SWAP m@ =
 7 AND NOT ABORT" Nonconformable " (TRNSP) ;
 8 ( Exchanging Rows of Interlaced Matrix, e.g. ' X ' Y 3 2 excR )
 9 ~ CODE excR 2 POP 0 POP 1 POP W POP I PUSH U PUSH
10 1 I MOV W ) 0 ADD 65528 # U MOV
11 I ) 2 ADD 4 W) 1 MOV 2 W) W MOV 65520 # I MOV BEGIN 0 DS LSG
12 R64 I ) FLD R64 3 ) FLD 2 DS LSG R64 I ) FLD R64 3 ) FLD
13 0 DS LSG R64 3 ) FSTP R64 I ) FSTP 2 DS LSG R64 3 ) FSTP
14 R64 I ) FSTP W 0 ADD W 2 ADD LOOP 2 2 SUB 2 DS LSG
15 U POP I POP NEXT
```

```
219 LIST

 0 ( Matrix Multiplication, 10th-.099 sec. 50th- 9.817 sec. )
 1 CODE (A*) I PUSH R PUSH U PUSH 65520 # I MOV
 2 a 2 MOV k R MOV R ES LSG c 0 MOV m 1 MOV 1 R MOV BEGIN 1 PUSH
 3 b W MOV n 1 MOV BEGIN 1 PUSH 2 U MOV FLDZ
 4 1 ES SSG BEGIN U DS LSG R64 I ) FLD W DS LSG W INC
 5 R64 I ) FMUL R U ADD 1 N) FADD LOOP
 6 0 DS LSG R64 I ) FSTP R 0 ADD 1 POP LOOP
 7 2 INC 1 DS LSG c LDA 0 INC c STA 1 POP LOOP
 8 1 ES LSG U POP R POP I POP NEXT
 9 : A* c! b! a! c@ dim@ n! m! adr@ c! b@ dim@ n@ = SWAP k!
10 adr@ b! a@ dim@ k@ = SWAP m@ = adr@ a! AND AND NOT
11 ABORT" Nonconformable " (A*) ;
12
13
14
15
```

```
220 LIST

  0 ( Subroutines for Matrix Inversion )
  1 ~ CODE (INV1) I PUSH n LDA 0 1 MOV j W MOV
  2 W MUL adr W ADD W 0 ADD 0 DS LSG 65520 # I MOV  R64 I ) FLD
  3 FLD1 1 N) FDIV REV W 2 MOV 65528 # W MOV 1 0 MOV BEGIN 0 N) FLD
  4 2 DS LSG R64 I ) FMUL R64 I ) FSTP 0 N) FLD R64 W ) FMUL R64
  5 W ) FSTP 0 2 ADD LOOP 2 2 SUB 2 DS LSG I POP  0 N) FSTP NEXT
  6 ~ CODE (INV2) I PUSH R PUSH U PUSH 65520 # I MOV 65528
  7 # W MOV n LDA 0 1 MOV i MUL adr 0 ADD 0 R MOV i 0 ADD b STA
  8 adr LDA 0 ES LSG 0 # j MOV BEGIN 1 PUSH R 1 MOV b 1 SUB 0= NOT
  9 IF R DS LSG R64 I ) FLD FCHS 1 1 SUB 1 DS LSG i LDA k STA
 10 U ES SSG U 2 MOV j 2 ADD k U ADD n LDA 0 1 MOV BEGIN  0 N)
 11 FLD U DS LSG R64 I ) FMUL 2 DS LSG R64 I ) FADD R64 I ) FSTP
 12 0 N) FLD U DS LSG R64 W ) FMUL 2 DS LSG R64 W ) FADD 0 2 ADD
 13 R64 W ) FSTP 0 U ADD LOOP  0 N) FSTP THEN 1 1 SUB 1 DS LSG
 14 R INC j LDA 0 INC j STA 1 POP LOOP 1 1 SUB
 15 1 ES LSG U POP R POP I POP NEXT


221 LIST

  0 ( Matrix Inversion, 10th-.217 sec., 50th- 22.168 sec. )
  1 : INV a! a@ <square> n@ m! (I2!) n@ 0 DO n@ I j! j@ DO I k! I
  2       j@ AA@ F0= NOT IF LEAVE THEN LOOP a@ a@ j@ k@ excR
  3       (INV1) I i! (INV2) LOOP a@ a@ 2 1 -> ;
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15


222 LIST

  0 ( Determinant, .074 sec.-10th, 4.965 sec.-50th Order )
  1
  2 CODE (DET) I PUSH R PUSH U PUSH n 1 MOV k LDA 0 R MOV
  3 65520 # I MOV 0 1 SUB n MUL n 2 MOV adr 0 ADD 0 U MOV
  4 R 0 ADD j U ADD BEGIN 0 N) FLD 0 DS LSG R64 I ) FMUL
  5 U DS LSG R64 I ) FADD R64 I ) FSTP 2 0 ADD 2 U ADD LOOP 0 N)
  6 FSTP 2 2 SUB 2 DS LSG U POP R POP I POP NEXT
  7 : D1 k@ k@ AA@ FNEGATE 1/N n@ k@ 1+ DO FDUP I k@ AA@ F* I j!
  8 (DET) LOOP FDROP ;
  9 : D2 n@ k@ 1+ DO I k@ AA@ F0= NOT IF n@ k@ DO J I AA@ k@ I AA@
 10 J I AA! k@ I AA! LOOP b@ NEGATE b! D1 LEAVE THEN LOOP ;
 11 : DET a! a@ <square> a@ a@ 1 2 -> 1 b! n@ 1- 0 DO I k! I I
```

(Continued on next page)

Fast Matrix Operations

(Listing continued, text begins on page 70)

```
 12 AA@ F0= NOT IF D1 ELSE D2 THEN LOOP 1.0 n@ 0 DO I I AA@ F* LOOP
 13 b@ >N F* a@ a@ 2 i -> ;
 14
 15
```

```
223 LIST

  0 ( Column Sort, 500 Numbers-.740 sec. )
  1 CODE (SORT) I PUSH R PUSH U PUSH 65520 # I MOV a 2 MOV b U MOV
  2 2 ES LSG BEGIN U SHR U U AND  0= NOT IF a! n 1 MOV U 1 SUB
  3 0 0 SUB j STA BEGIN j R MOV BEGIN 2 ES SSG R 2 ADD 2 W MOV U W
  4 ADD 2 DS LSG  R64 I ) FLD W DS LSG  R64 I ) FLD 1 N) FCOM
  5 0 0 SUB 0 DS LSG a FSTSW a LDA SAHF CS IF 2 DS LSG R64 I ) FSTP
  6 W DS LSG R64 I ) FSTP U R SUB 0< IF 0 0 SUB THEN ELSE
  7 0 N) FSTP 0 0 SUB  0 N) FSTP THEN SAHF CS NOT UNTIL 0 0
  8 SUB 0 DS LSG j LDA 0 INC j STA 1 0 CMP 0> UNTIL 1 0 CMP 0<
  9 UNTIL a@ THEN 0 0 SUB 0 DS LSG 0 ES LSG U POP R POP I POP NEXT
 10 ~ : sortC1 a! dim@ SWAP m! m@ b! b@ 1- n! a@ * adr@ + a!
 11 (SORT) ;
 12 ~ : sortA1 dim@ n! m! n@ m@ * b! b@ 1- n! adr@ a! (SORT) ;
 13
 14
 15
```

```
224 LIST

  0 ( Column Sort, 500 Numbers-1.0 sec. )
  1 CODE (SRT) I PUSH R PUSH U PUSH 65520 # I MOV a 2 MOV b U MOV
  2 2 ES LSG BEGIN U SHR U U AND  0= NOT IF a! n 1 MOV U 1 SUB
  3 0 0 SUB j STA BEGIN j R MOV BEGIN 2 ES SSG R 2 ADD 2 W MOV U W
  4 ADD 2 DS LSG  R64 I ) FLD W DS LSG  R64 I ) FLD 1 N) FCOM
  5 0 0 SUB 0 DS LSG a FSTSW a LDA SAHF CS IF 2 DS LSG R64 I ) FSTP
  6 W DS LSG R64 I ) FSTP R64 8 I) FLD 2 DS LSG R64 8 I) FLD W DS
  7 LSG R64 8 I) FSTP 2 DS LSG R64 8 I) FSTP U R SUB 0< IF 0 0 SUB
  8 THEN ELSE 0 N) FSTP 0 0 SUB  0 N) FSTP THEN SAHF CS NOT UNTIL
  9 0 0 SUB 0 DS LSG j LDA 0 INC j STA 1 0 CMP 0> UNTIL 1 0 CMP 0<
 10 UNTIL a@ THEN 0 0 SUB 0 DS LSG 0 ES LSG U POP R POP I POP NEXT
 11 ~ : sortC a! dim@ SWAP m! m@ b! b@ 1- n! a@ * adr@ + a!
 12 (SRT) ;
 13 ~ : sortA dim@ n! m! n@ m@ * b! b@ 1- n! adr@ a! (SRT) ;
 14
 15
```

```
225 LIST

 0 ~ CODE polyR1 0 POP W POP W ) 0 ADD 4 W) 1 MOV 2 W) 2 MOV
 1 65520 # W MOV FLD1 1 DEC 0 DS LSG R64 W ) FST BEGIN
 2 1 N) FMUL NO 2 0 ADD 0 DS LSG R64 W ) FST LOOP 0 N) FSTP
 3 0 N) FSTP 2 2 SUB 2 DS LSG NEXT
 4 ~ CODE polyV1 W POP W ) 0 MOV  2 W) 1 MOV
 5 65520 # W MOV FLD1 1 DEC 0 DS LSG R64 W ) FST BEGIN
 6 1 N) FMUL NO 0 INC 0 DS LSG R64 W ) FST LOOP 0 N) FSTP
 7 0 N) FSTP 2 2 SUB 2 DS LSG NEXT
 8 CODE V* W POP W ) 0 MOV  2 W) 1 MOV W POP W ) W MOV
 9 I PUSH 65520 # I MOV FLDZ BEGIN 0 DS LSG R64 I ) FLD W DS LSG
10 R64 I ) FMUL 0 INC  W INC 1 N) FADD LOOP 0 0 SUB 0 DS LSG
11 I POP NEXT
12
13
14
15


226 LIST

 0 ( Matrix Transpose-Multiply, 10th-.093 sec. 50th- 9.818 sec. )
 1 CODE (T*) I PUSH R PUSH U PUSH k R MOV b 2 MOV c 0 MOV
 2 65520 # I MOV n 1 MOV BEGIN 1 PUSH adr W MOV m 1 MOV
 3 BEGIN 1 ES LSG 2 U MOV FLDZ R 1 MOV BEGIN U DS LSG R64 I ) FLD
 4 W DS LSG R64 I ) FMUL W INC U INC 1 N) FADD LOOP 0 DS LSG
 5 R64 I ) FSTP 0 INC 1 ES SSG LOOP 1 DS LSG R 2 ADD 1 POP
 6 LOOP 1 DS LSG 1 ES LSG U POP R POP I POP NEXT
 7 : T* dim@ n! m! adr@ c! dim@ n@ adr@ b! = SWAP k! SWAP dim@ m@
 8 = SWAP k@ = AND AND NOT ABORT" Nonconformable " (T*) ;
 9
10
11
12
13
14
15


227 LIST

 0 ~ CODE (polyA1) I PUSH R PUSH 65520 # I MOV m LDA n 2 MOV 2 DEC
 1 adr W MOV 0 1 MOV BEGIN 1 ES LSG W R MOV R DS LSG R64 I ) FLD
 2 FLD1 R64 I ) FST 2 1 MOV BEGIN 0 R ADD R DS LSG 1 N) FMUL NO
 3 R64 I ) FST LOOP 0 N) FSTP 0 N) FSTP W INC 1 ES SSG LOOP
 4 1 ES LSG 1 DS LSG R POP I POP NEXT
 5 ~ : polyA1 dim@ n! m! (polyA1) ;
 6
 7
 8
 9
10
11
12
13
14
15
```

End Listing

DR. DOBB'S CLINIC

by D. E. Cortesi, Resident Intern

Two Gets You Ten

Some time back we posed a question regarding binary floating point versus decimal floating point. There are some ratios (fractions) that can be represented by a finite number of decimal digits, but which require an infinite number of binary digits. For example, the fraction 1/5 can be represented exactly in decimal (as 0.2), but it becomes a "repeating binary" when you attempt to represent it as a binary fraction (namely, 0.00110011...). It takes an infinite number of bits to represent *exactly* 1/5 in binary. Since any machine representation is woefully finite, it is quite literally impossible to store a value of 1/5 in a system that uses binary float. You may enter 0.2, but what is stored is 0.199..., for however many effective digits the system supports.

Ah yes, the question. There are ratios that have a finite representation in decimal but not in binary; we asked if there were also ratios that have a finite representation in binary but not in decimal? David Oster wrote to say that there were not, and Robert Pirko provided the following amplification.

"David Oster is correct that all fractions which have a finite representation in binary also have one in decimal. To see why, note that any finite-length decimal number can be written as

$$d = \frac{p}{10^n} \qquad \text{(eq. 1)}$$

where p and n are integers. For example,

$0.35 = 35/10^2$
$0.357 = 357/10^3$

"Similarly, a finite-length binary number has the form

$$b = \frac{q}{2^m} \qquad \text{(eq. 2)}$$

where q and m are integers. For example,

$0.101 = 5/2^3 = 0.625$ decimal
$0.1011 = 11/2^4 = 0.6875$ decimal

We can multiply the top and bottom of equation 2 by a constant of 5^m;

$$b = \frac{q * 5^m}{2^m * 5^m} = \frac{q * 5^m}{10^m} \qquad \text{(eq. 3)}$$

but this is equivalent to equation 1, so the value b can thus be precisely specified in m decimal digits."

Pirko goes on to note an interesting implication of this. That the denominator of equation 3 is 10^m implies that it could take as many as m decimal digits to represent an m-bit binary fraction in decimal. If there are 56 bits in the fraction (as in an MBASIC double-precision number), some representable values will require 56 digits for exact representation in decimal – but no decimal-display routine will show them (as MBASIC only displays 16 significant digits).

We are beginning to see a little more clearly the relationship between binary and BCD floating-point machine arithmetic. The problems unique to binary arise during input and output conversions. On input, certain decimal fractions are truncated (1/5 becoming 0.199...). On output, certain binary values are truncated. A representation using BCD will suffer neither of these conversion errors.

Is either system significantly better during the arithmetic operations that happen between input and output? Mathematically speaking, we suspect not. All operations will be done on a finite number of digits (whether binary ones or decimal ones); therefore either method will incur roundoff errors and all the peculiarities they induce. Practically speaking, the errors might differ somehow in quality and so affect a user's *perception* of a system's accuracy. Comments, whether theoretical or based on experience, are still welcome on this point. We ask again, has anybody run comparable problems on a pair of comparable systems (e.g., binary BASCOM versus decimal CB80)?

If you are having trouble with binary arithmetic, you should note a letter from Paul Weras, published in the May 9 edition of *InfoWorld*. He suggests that the PRINT USING statement of BASIC will fix most of the problems people have with money output, giving as an example

```
10 FMT$ = "#.##"
20 A = 9.899999
30 PRINT USING FMT$; A
```

which results in a display of 9.90 – the numeric format item apparently having caused rounding.

Cheap Thrills Dept.

We are always happy to hear about good, cheap software, so we were pleased to get a note from Jim Laurino of Philadelphia, Pennsylvania. He writes a wholehearted recommendation for PE, a full-screen editor for the IBM PC. "It has unique features," he says, "that place it head and shoulders above similar products. IBM, which sells it under part number 6024051, doesn't seem to appreciate it. The sales people I have spoken to had no experience with it. Perhaps because it is priced at only $100, people *assume it's* not very good."

One of PE's best features, it seems, is that it allows you to program the values of all 99 key operations, giving any alt-, ctl-, shift, or normal stroke a value that can apparently be a single character or a lengthy string including mixed data and editor operations. The key definitions are read from a "pe.pro" file when the editor starts up, and the active definitions can be edited as if they were a file during the edit session. PE can also edit the disk directory as if it were a file, and Laurino gives some keystroke definitions that allow him to load the disk directory and convert it into a batch command file with a half-dozen keystrokes.

"You can have up to 20 files open at once," Laurino says, "including directories and internal files for key definitions and changed lines. You may mark text in any file and copy it to any other. If you have only two files open, typing F8 rapidly will switch you between them so quickly that differences appear to flicker, giving you a visual comparison function. PE knows about and can use all the memory you have as a buffer, and will overflow to disk when it runs out."

The AGGHHH! Program

Our CP/M Plus Bios is still a wee bit shaky and every once in a while it takes off to never-never land. Alas, this usually happens when we are in the middle of editing. The first time we lost the system after an hour's typing, we got the file back by loading up SID, prowling around the shattered remains of the PeachText edit buffer, and finally hand-assembling a little program that wrote the still-readable text to disk as a file.

The second time it happened, we got serious and wrote the Panic program. Panic will recover whatever remains in storage of a PeachText file after a system crash. Those who use PeachText version 2 might be interested in it (Listing, page 81).

The text buffer in storage begins with a byte of FFh and ends with another

FFh. Between the two, there should be nothing but ASCII data. There may be a large number of NUL (00h) bytes; these are not saved. Block markers seem to be represented as 01h. There are no tabs; they are apparently expanded to blanks when the file is loaded. The CR is a line-marker but it is not followed by a LF, so Panic has to insert one. All this should also be true of earlier versions of PeachText (Magic Wand), but the address "Peaches," where Panic expects to find the leading FFh, would probably be different. ▶▶J

(Listing begins below)

Clinic Listing

```
                ;================================================================
                ;
                ;       PANIC  fileref
                ; Save the contents of a crashed Magic Wand's edit buffer
                ;
                ; The buffer should begin at 41F8 (PeachText) with a byte of FF.
                ; Data follows to a second FF.  The data should be all-ASCII but
                ; it may include control characters:
                ;   CR (0Dh) - a LF (0Ah) must be added to each one
                ;   LF (0Ah) - may appear and is OK
                ;   FF (0Ch) - may appear ("page mark") and is ok
                ;  TAB (09h) - does NOT appear.
                ;  SOH (01h) - seems to be a "block marker" -- discard
                ;  NUL (00h) - sometimes appears -- discard
                ;
                        CSEG
0000 314000             lxi     sp,Stack
0003 C37500             jmp     Main
                ;
000D =          CR      equ     0Dh
000A =          LF      equ     0Ah
0020 =          Blank   equ     20h
0024 =          Dollar  equ     '$'
001A =          Eof     equ     26
41F8 =          Peaches equ     41F8h   ; where the edit buffer should be
005C =          Fcb     equ     5Ch     ; a fileref MUST BE SUPPLIED
0080 =          Record  equ     0080h   ; file buffer for output
0005 =          Bdos    equ     5       ; Bdos vector
0009 =          TypeMsg equ     9       ; type a string to $
0013 =          Erase   equ     19      ; erase function
0016 =          Make    equ     22      ; make new file
0015 =          Write   equ     21      ; sequential write
0010 =          Close   equ     16      ; close output file
001A =          SetAd   equ     26      ; set file buffer address
                ;
0006 4669727374NoFF     db      'First FF not found',CR,LF,Dollar
001B 5365636F6ENoEnd    db      'Second FF not found',CR,LF,Dollar
0031 5573616765NoFile   db      'Usage is: PANIC filespec',CR,LF,Dollar
004C 536F6D6520Done     db      'Some data saved, use with care',CR,LF,Dollar
                ;
                Quit: ; come here with DE-> one of above messages, to quit
006D 0E09               mvi     c,TypeMsg
006F CD0500             call    Bdos
0072 C30000             jmp     0
                ;
                Main:
                ;
                ; First off, make sure we have a starting FFh
                ;
```

(Continued on next page)

Clinic Listing

(Listing continued, text begins on page 80)

```
0075 21F841             lxi     h,Peaches
0078 7E                 mov     a,m
0079 3C                 inr     a       ; is it FF (goes to 00)?
007A CA8300             jz      Main2   ; (yes, fine)
007D 110600             lxi     d,NoFF  ; no -- PeachText overlaid?
0080 C36D00             jmp     Quit
                ;
                ; Then make sure we have an ending FF at a higher address,
                ; but still less than the start of the Bdos
                ;
0083 3A0700     Main2:  lda     Bdos+2  ; page-address of Bdos
0086 47                 mov     b,a     ; to B for comparisons
0087 111B00             lxi     d,NoEnd ; message for if we quit
                Main3:
008A 23                 inx     h
008B 7C                 mov     a,h
008C B8                 cmp     b       ; still below Bdos?
008D D26D00             jnc     Quit    ; (no, give up)
0090 7E                 mov     a,m
0091 3C                 inr     a       ; found the 2nd FF yet?
0092 CA9B00             jz      Main4   ; (yes, carry on)
0095 FA6D00             jm      Quit    ; (no, found garbage (code?))
0098 C38A00             jmp     Main3   ; (no, but still in Ascii data)
                ;
                ; Now make sure that we have a filespec to work with
                ;
009B 3A5D00     Main4:  lda     Fcb+1
009E FE20               cpi     Blank   ; omitted first operand?
00A0 113100             lxi     d,NoFile
00A3 CA6D00             jz      Quit    ; (yes, bail out to try again)
                ;
                ; Get rid of any existing file of this name (well, this IS
                ; a crisis, you know)
                ;
00A6 115C00             lxi     d,Fcb
00A9 0E13               mvi     c,Erase ; erase
00AB CD0500             call    Bdos    ; ..the named file
00AE 115C00             lxi     d,Fcb
00B1 0E16               mvi     c,Make  ; make
00B3 CD0500             call    Bdos    ; ..the named file
00B6 118000             lxi     d,Record
00B9 0E1A               mvi     c,SetAd ; set
00BB CD0500             call    Bdos    ; ..the file buffer address
                ;
                ; set up registers for the loop and get to work
                ;
00BE 118000             lxi     d,Record ; DE->next outbyte
00C1 21F941             lxi     h,Peaches+1 ; HL->next inbyte
                ;
00C4 7E         loop:   mov     a,m     ; get this byte,
00C5 23                 inx     h       ; ..and point to next one
00C6 B7                 ora     a       ; is this FF?
```

```
00C7 FAF200             jm      closeit ; (yes, finish up)
00CA FE0A               cpi     LF      ; is it less than LF?
00CC DAC400             jc      loop    ; (yes, ignore 00..09)
00CF CDDC00             call    onebyte ; whatever, write it
00D2 FE0D               cpi     CR      ; CRs need LFs in files
00D4 3E0A               mvi     a,LF
00D6 CCDC00             cz      onebyte ; add LF to every CR
00D9 C3C400             jmp     loop    ; and repeat
                ;
                ; here write the byte in A, sending a record to disk
                ; as necessary.  Preserve current byte in A.
                ;
                onebyte:
00DC 12                 stax    d       ; store the byte
00DD 1C                 inr     e       ; step the pointer
00DE CCE200             cz      writit  ; (write block if full)
00E1 C9                 ret
                ;
                ; send one record to disk, reset outpointer in DE,
                ; preserve A, HL
                ;
                writit:
00E2 F5                 push    psw     ; block full, save byte
00E3 E5                 push    h       ; ..and HL
00E4 115C00             lxi     d,Fcb
00E7 0E15               mvi     c,Write ; and write
00E9 CD0500             call    Bdos    ; one block
00EC 118000             lxi     d,Record; reset outpointer
00EF E1                 pop     h       ; recover text pointer
00F0 F1                 pop     psw     ; and byte
00F1 C9                 ret
                ;
                ; finish up the saved file by closing it
                ;
                closeit:
00F2 3E0D               mvi     a,CR    ; tack on a newline
00F4 CDDC00             call    onebyte
00F7 3E0A               mvi     a,LF    ; just to be safe
00F9 CDDC00             call    onebyte
00FC 3E1A               mvi     a,Eof   ; and an eof
00FE CDDC00             call    onebyte
0101 1D                 dcr     e       ; if we didn't just start a record
0102 FCE200             cm      writit  ; (ie, 80 goes to 7F) then write last.
0105 115C00             lxi     d,Fcb
0108 0E10               mvi     c,Close
010A CD0500             call    Bdos
010D 114C00             lxi     d,Done  ; reassure user
0110 C36D00             jmp     Quit    ; and exit
                ;
                        DSEG
0000                    ds      64
                stack:
0040                    end
```

End Listing

CP/M EXCHANGE

by Robert Blum

This issue marks the first anniversary of the CP/M Exchange. The intent of this column is the same today as it was one year ago: to provide a forum for CP/M users to share their experiences. It makes no difference whether you are a first time CP/M user or have been using it for many years, the CP/M Exchange is your column. If you have questions send them along. I will try to answer them myself or seek counsel from the other readers. On the other hand, your experiences and solutions to problems can benefit others. It's been an exciting year. Numerous subjects have been reviewed and your response has been gratifying. Here's to you and the next twelve months.

In Retrospect

The most popular subject covered in the past year was communications. Gene Head's MBOOT program was greeted with a flurry of mail; and even several months later, I still receive an occasional letter. In reviewing all the mail addressed to communications, it's obvious that going online for the first time can be a frustrating experience. In a few letters it was made clear that, if a significant investment in hardware had not been made, the whole project would have been scrapped. Although, once the problems were worked out, the experience of being able to dial into systems around the country and peruse the myriad pieces of free software made the effort worthwhile.

Before attempting to dial into a remote system for the first time, a little extra preparation is necessary. Keep in mind that the clock doesn't stop while you are figuring out a problem. Every minute saved on a coast-to-coast call can represent a substantial savings. First, study and restudy the documentation. It's important to understand what is about to happen and the responses to any error messages. Second, make a command reference card. This card will become your best friend the first time you can't remember how to respond to a message. Third, take advantage of the many long distance phone services that offer reduced rates. If no discount phone service is available in your area, wait for weekends or evenings when rates are reduced. Last, try to find a local system to experiment with before going to the long lines. It's important to verify that your system is operational before committing yourself to long distance phone bills.

MBOOT Revisited

Dr. David DuPuy, of Lexington, Virginia, wrote to share his experiences and changes to MBOOT (see Figure 1, page 87) for the Morrow MULT/IO board. He writes:

"Your section on using XMODEM was very useful; I had no idea of the protocol with XMODEM. It would be very useful to many of your readers to give the sequence required to 'unsqueeze' a file. I had a little difficulty in locating the unsqueeze program and learning the proper commands to run the program. The name of the file on the RCP/M I used (Ratoff's) was USQ-15.OBJ. After downloading this file, rename it to USQ-15.COM and then run it with 'USQ-15 PROG.AQM'."

Dr. DuPuy goes on to say that he has started work on converting MBOOT to run with the CompuPro Interfacer 4 I/O board but would appreciate the code if somebody else has already completed it. Finally, he suggests that a control-P trap be implemented to enable online printing of everything that goes across the line.

CP/M Plus Feedback

Victor Ung, of Monterey Park, California, writes about his experiences on implementing CP/M Plus:

"The following is intended to be of assistance to those who are trying to generate a CP/M Plus system. These tips apply if you are writing a BIOS from scratch as outlined in the CP/M Plus *System Guide*. I assume that you have purchased a generic system and that you have thoroughly read the documentation.

"Z80 macros versus the real thing. I don't understand why people can't program a Z80 without inventing their own opcodes. If you wish, you can use Microsoft's MACRO-80 assembler to generate .REL files instead of using the Z80.LIB macros supplied with RMAC.

"Most of the problems I had with the CPMLDR were because of the *System Guide's* simple understatement that 'the (CPMLDR) LDRBIOS is essentially a nonbanked (CP/M Plus) BIOS.' This prompted me at first to try to link my nonbanked version of BIOS3 with CPMLDR.REL which led me to diagnose the following problems:

"When CPMLDR calls the BOOT routine of LDRBIOS, it does so with a small eight-level stack. Only three levels are left before the delimiter of the still undisplayed sign-on message is overlaid. If the BOOT routine overflows the stack, CPMLDR will not only display the DRI copyright but will also treat you to an ASCII memory dump of your system.

"CPMLDR needs a non-relocatable SCB. In LDRBIOS you should define the SCB as an absolute block of storage. Otherwise the SCB will be located at FE00H, where it will be overlaid as CPM3.SYS is loaded.

"There is no automatic disk buffer allocation in CPMLDR. You must define all values for the DPH and DPB of the drive used to load the system. A directory, DIRBCB, and data buffer, DATABCB, must also be allocated.

"The CP/M BIOS is where the improved sysgen documentation really shines. Everything you need to know about your new BIOS is in the *System Guide*. However, if you are trying to generate a banked version of BIOS3, referred to as BNKBIOS3, then you should carefully read section 3.5 on 'Banking Considerations.'

"A possible RMAC bug may exist if you use the CPM3.LIB macro library to define large drive characteristics. While I am not ready to cry BUG! yet, I have had some problems using the DPB macro to define large (over 5MB) drives. Carefully examine the PRN output file and if you notice wrong values for DSM, DRM, and CKS, then you should give up and manually calculate and code the proper DPB values.

"It is well worth the effort to write a new BIOS from the ground up rather than to hack up an existing CP/M 2.2 BIOS. It took me about 30 hours to get a nonbanked system up and another four hours to convert it to a banked system. The chapter on how to debug the BIOS is excellent and I had no problem understanding my mistakes. DRI could, however, improve section 5.2 on 'Customizing the CPMLDR.'"

Searching Further

Several months ago I asked for help in locating a CCP patch that will search user area 0 for a .COM file not found in the current user area. Ernest Payne, of Tucson, Arizona, responded in grand style. He sent along a complete disassembler source listing of the CCP and BDOS for CP/M 2.2. He also pointed me to an article in the March/April 1982 issue of

```
;     From DDJ Oct '82 page 47
;     To serve as simple terminal/modem program....
;
;     First, my equates (DLD)
;
false     equ  0
true      equ  not false
stdcpm    equ  true
exitchr   equ  05h
esc       equ  1bh
fastclk   equ  true       ;work with 6Mhz clock
;
modctlp   equ  4dh        ;modem control port
modsndb   equ  20h        ;my bit to test to send
modsndr   equ  20h        ;my bit to say ready to send
modrcvb   equ  01h        ;my bit to test to receive
modrcvr   equ  01h        ;my value when ready to rcvr
moddatp   equ  48h        ;my modem data port
;
errlim    equ  10
soh       equ  1

288:          if   fastclk
289:          mov  a,b
290:          add  a
291: ;    now for 6Mhz clock
292:          add  b          ;this statement added for 6Mhz
293:          mov  b,a
294:          endif
329: ;
330: ;    initialization for Morrow MULT/IO board
331: base     equ  48h        ;basic port address
332: initmod  mvi  a,2        ;group select (ACE #2)
333:          out  base+7     ;output it
334:          mvi  a,135      ;set DLAB, 8 bit data, 2 stop
335:          out  base+3     ;output that
336:          mvi  a,80h      ;low byte BAUD RATE
337:          out base        ;output that
338:          mvi  a,1        ;high byte BAUD RATE
339:          out  base+1     ;output that for 300 BAUD
340:          mvi  a,7        ;clear DLAB
341:          out  base+3     ;output to line control register
342:          xra  a          ;clear A
343:          out  base+5     ;clear data available flag
344:          mvi  a,0        ;disable interrupts
345:          out  base+1     ;output that
346: ;
347:          mvi  a,2        ;be sure group select is 2
348:          out  base+7     ;output that
349:          mvi  a,7        ;be sure DLAB is low
350:          out  base+3     ;output that, and that's all.
351: ;
352:          ret
```

Figure 1.
MBOOT Update for Morrow MULT/IO Board

Microsystems magazine which contains the necessary BIOS changes to implement this enhancement. If you don't have access to this issue, send me a SASE and I will return a copy of it to you.

If you don't have access to the BIOS source code for your system, or you want to go beyond this one change, a complete Z80 CCP replacement is available from SIG/M. I have been using the original version of ZCPR for over a year now and have become so accustomed to its extra features that I have a difficult time using the standard version of CP/M's CCP. The latest version of ZCPR (ZCPR2) is contained on SIG/M volumes 98 through 107. No, I didn't make a mistake; it takes ten 8" disks to hold all the source code and documentation plus a few other goodies. For a brief description of ZCPR's non-standard functions, consult Figure 2 (below left). SIG/M can be reached at Box 97, Iselin, New Jersey 08830.

(A) Z80 code is used throughout to reduce the size of the CCP and give room to implement the additional functions. Also, the input line buffer has been reduced in size to approximately 80H bytes.

(B) A new Command-Search hierarchy has been implemented:
1. Scan the list of built-in CCP commands.
2. If the command is not CCP-resident, look for a .COM file on the current disk in the current user area.
3. If the .COM file is not found in the current user area and the current user area is not the default user area (normally 0), then the default user area is selected and scanned for the file.
4. If the .COM file is not found on the currently logged drive, drive A: is selected and scanned for the file under the default user area.

(C) The default user number can be changed by the DFU command.

(D) A new command called PAGE will output a file to the CON: device one screen-full at a time and wait for a continue character. Otherwise, it is like TYPE.

(E) The new LIST command sends a file to the LST: device.

(F) The CLS command clears the terminal screen.

(G) The user number is printed as part of the command prompt; the prompt is now du> (e.g., "A0>" or "A15>").

(H) The ERA command now displays the names of all the files it has erased.

(I) The DIR command has the additional special form "DIRA" which displays all files (both non-system and system), while "DIR" displays just the non-system files.

(J) The directory display no longer displays the disk name at the beginning of each line and it now includes a "." between the file name and file type (e.g., FILE.TYP).

(K) The SUBMIT File Facility now expects the $$$.SUB file to be on the currently logged-in disk (as opposed to always A:).

(L) The command line prompt is now "=" if the command comes from a $$$.SUB file and ">" if the command comes from the user. Also, the ">" is not printed until all preprocessing is completed.

(M) A form feed is sent to the LST: device by the FF command.

(N) The new LF command sends a variable number of line feeds to the LST: device.

(O) All CCP commands (such as SAVE), which normally take a decimal number, now will also accept hex numbers (e.g., 20H).

(P) The new SAVS command is like SAVE except that it saves a number of disk sectors.

(Q) A pair of commands have been added to allow loading a file anywhere in memory (GET) and then executing it (JUMP xxx). Of course, JUMP could also execute a PROM program. GET prevents a file from overwriting BDOS or BIOS.

(R) A "GO" command has been added which does an immediate jump to 100H.

(S) Doing a warm boot now maintains the current user and drive numbers even when recovering from an I/O error.

(T) The HELP command displays a list of the resident commands.

Figure 2.
ZCPR Non-Standard Functions

Take note of one problem before modifying your BIOS or implementing ZCPR. Multiple-phase programs such as WordStar or dBase II that load overlays at run time may not always execute properly. Once the CCP has loaded the program from the proper user area and turned control over to it, the application program is unaware of which user area contains its overlay segments. No easily implemented solution for this inadequacy is available. The only sure way to avoid problems of this type is to restrict yourself to one user area or to copy multi-phase programs into all user areas where they will be used.

64K and Beyond

Memory management techniques have been in use for years on large computers. Their reason for existence is to expand the number of uniquely addressable memory positions beyond the limitations imposed by the cpu. 8080 and Z80 processors provide 16 address bits, thereby limiting the maximum number of directly addressable positions to 65,536. 64K of memory may seem adequate considering that only a few years ago most micros were operating with 16K. But as programs become more sophisticated, their appetite for memory increases. For example, one of the more popular spreadsheet programs provides 13K of free memory in a 64K CP/M system. Unfortunately this equates to only several hundred active cell positions – barely adequate for any type of serious financial modeling. Adding 4K to 6K of memory expands the maximum sheet size to the point of overtaxing a 4MHz Z80.

Going beyond 64K requires additional hardware that automatically inserts extra high-order address bits when a memory transfer takes place. Most of the the S-100 cpu and memory boards being manufactured today contain additional logic for this purpose.

In order to understand how extended address systems work, visualize one 16K fixed segment of memory with a base address of C000H and several 48K segments of memory each with a common base address of 0000H. All memory segments will respond to the normal 16 bits of address information presented by the cpu. Additionally, each 48K banked memory segment will also respond to 8 bits of extended address information. When a memory transfer is to take place on the bus, the cpu puts out its 16 bits of address information to select one byte position

within one of the 48K segments. At the same time, the extended address logic asserts its 8 bits of extended address information to select one of the 48K banked segments. Hence, one byte within one of the 48K segments is selected. Reviewing Figure 3 (at right) at this time may be helpful.

Two basic types of memory expansion are employed today. Conceptually they are the same; their difference exists in how they are implemented. The oldest method, Cromemco standard, requires that each memory board have an I/O port at address 40H to receive a one-byte selection code, and some form of static switch to preset a unique bit pattern used as the extended address. A simple output instruction to port 40H will then enable the memory bank with the matching preselected bit pattern and disable all others. The newest method, IEEE-696 S-100 bus standard, sets aside 8 extra bus address lines for use as an extended address. When used in conjunction with the normal 16 address lines, this effectively provides a 24-bit address. Each memory board in this type of system must have enough on-board selection logic to properly respond to all 24 bits of address for any memory transfer. The drivers for these 8 extended address lines are usually a feature of the cpu board. Again, only one output instruction is needed to select the desired banked memory segment.

Some software coordination is also necessary to allow this type of system to operate properly. Any information transfer between memory banks must be passed through the fixed, or common, bank of memory because only one bank can be enabled at a time. This task is usually delegated to a special subroutine of CP/M's BIOS.

I think you will agree that, once some of the basic principles of bank selection are understood, CP/M Plus's operation will become more readily apparent. Next month I will continue with a wrap-up of CP/M Plus's features. In following months I will be discussing the steps necessary to bring CP/M Plus up and any problems that I encounter. I'm looking forward to this adventure because I purchased a 256K dynamic memory board just so I can put all the new features to use. ■■J

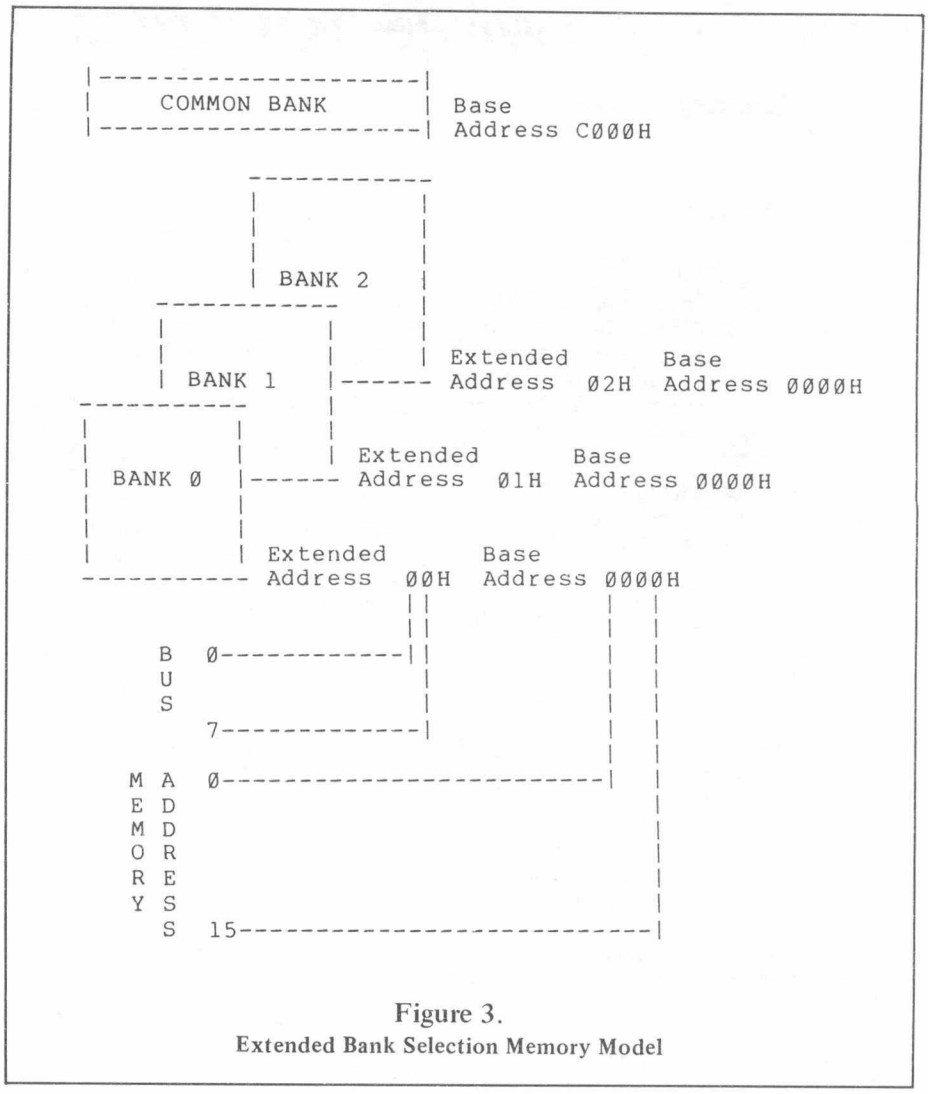


Figure 3.
Extended Bank Selection Memory Model

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

MS-DOS Rebuttal

Tim Paterson, the author of MS-DOS 1.0 and currently director of engineering at Seattle Computer, wrote in to respond to some of my comments about PC-DOS in the May 1983 column. The following paragraphs are taken verbatim from the body of his letter:

"Neither CP/M-86 nor MS-DOS check if a disk has been changed when a file is read or written – only when it is opened or closed. Such a test would be totally impractical. In the CP/M-86 case, files are written 128 bytes at a time by making successive calls to the operating system. To test for a changed disk would require a read of the disk directory before writing each sector. If the test is to be made at all, it must be made on every 'write sector' system call, since the operating system has no way to know when it needs to check and when it doesn't. The extra overhead involved with seek time, latency delays, and reading from the directory would increase the time to write a file by at least an order of magnitude.

"The way both operating systems work is like this: Both CP/M-86 and MS-DOS keep a map in memory of available disk space. Each entry in the map corresponds to a chunk of free space of fixed size, such as 1 Kbyte. As more space is needed to extend a file, the map is searched for free space to be assigned to the file. That new chunk of space is then written to as required for the file. If the disk has been changed since the file was opened, the map of free space will be incorrect, and the data in another file may get destroyed at this point.

"When the file is closed, the two systems diverge only slightly. CP/M-86 performs a checksum of the directory, which it compares with a stored value from a previous checksum. If the checksum differs, the disk is marked 'read-only' and the close fails. MS-DOS searches for the file in the directory and, if found, compares its position in the directory with *its position* when it was opened, which is stored in the File Control Block. If these do not match, the close fails and internal buffers (including the free space map) are invalidated.

"Of course, it is critical that the map of free space in memory be kept up to date as disks are changed. CP/M-86 will only update its map when the operator types control-C, or with a 'disk reset' system call, even if it has detected a disk change. MS-DOS updates its map from the disk every time a file is opened, unless another file is still in the process of being written.

"MS-DOS allows disks to be changed whenever files are not open for writing. If application programs kept this to a minimum, then MS-DOS could be nearly bullet-proof. The rule that should be used is that whenever a program is going to wait for user input, it should close output files, then re-open them later (with corresponding disk check change) if writing is to continue. Thus the user sees that any time the computer stops to wait for him, he can change disks.

"The MS-DOS editor EDLIN makes the serious mistake of leaving the file open for writing during the editing process. When I wrote EDLIN (in the summer of 1980), it never occurred to me that people would swap disks while editing. It must happen, I suppose, when someone gets so distracted that they forget what they are doing. Unfortunately, I'm afraid EDLIN is not the only program leaving output files open; I suspect this is common to many other editors, and other programs as well.

"I tested what happens when disks are swapped within EDLIN, to prove for myself that it works the way I wrote it. I edited a small (7K) file, but swapped disks before ending the session. EDLIN gave no indication of an error. Using DIR and CHKDSK commands, the disk I swapped in appeared to be unchanged. I typed several of the files from the disk, and they were unharmed. 7K of data must have been written somewhere on the disk, but I couldn't find it. It might well have been written in the disk's free space.

"In summary, CP/M-86 does not 'carefully protect' the integrity of files on a disk that has been changed at the wrong time. Nor is the behavior of MS-DOS 'inexcusable,' since it is nearly the same as CP/M-86. For a given application program, both operating systems are about equally susceptible to destroying data on a disk that has been swapped in at the wrong time."

I am grateful to Mr. Paterson for taking the time to send this explanation, and I was certainly in error in stating that CP/M-86 will not allow data to be written into the wrong place on an improperly swapped disk. But his letter leaves several questions unanswered.

First, the test of EDLIN was not very exhaustive. Presumably, when the editing session was ended, MS-DOS was still working from the previous disk's allocation map – so the fact that it did not write the 7K edited file over some other file's data on the new disk was purely good luck. Next, if MS-DOS was working properly, it should have returned a failure code to EDLIN when the file was closed, since the disk had been improperly swapped. Either EDLIN didn't get the error code or it didn't bother to report it – but which?

Finally, even if it is unclear just how it occurs, PC-DOS has definitely been observed on rare occasions to clobber the directory of a disk that was improperly swapped in while a file was open for write. Although this happened to me several times soon after I bought my IBM PC, I have not encountered it recently. (Was it a subtle bug in PC-DOS version 1.0 that was eliminated in 1.1; did I just unconsciously learn to avoid situations that caused it; or did it require some fairly unusual combination of circumstances that by coincidence hasn't recurred?

Attempts to recreate such an error intentionally have been unsuccessful so far; but, from the description provided by Mr. Paterson, at least one window to such an event seems to be open. If the swapped disk happened to have a file by the same name in the same directory position as the file that was opened for write on the previous disk, would PC-DOS recognize that the disk had been changed, or go ahead and write the old disk's allocation map and directory entry onto the new disk? Further input from readers and from Mr. Paterson on this subject is welcomed.

68000 Tools

Steve Passe, of Englewood, Colorado, sent in a pair of 68000 subroutines (see Listings 1 and 2, pages 94-98) and wrote: "This is the first submission of what I hope will be a long string of 68K tool donations on my part. I decided to start by converting your first two 8086 tools (ascbin and binasc), and tried to stick to the format of the originals as much as possible. I made them take full advantage of the 32-bit word size of the 68000; thus, the legal range for both routines is +2,147,483,647 through -2,147,483,648. This made them a little longer than their 8086 counterparts as overflow and carry conditions had to be taken into account.

"I operate a BBS type service which I encourage people interested in the 68000 series chips to call. The board is primarily dedicated to the C programming language, but I will soon be setting aside space in the file system for 68000 public domain code also. There is also a 'room' in the bulletin board system dedicated to info/discussion of the 68K. Inform me if and when these routines are to be published and I will make sure that the code for them is online by that time. The BBS is up 24 hours, 7 days a week at (303) 781-4937."

Sizing Memory on the IBM PC

Those of you with really *big* applications programs on the IBM PC will have noticed that the maximum memory size that can be set on the machine's sense switches is 544 Kbytes, even though according to the published memory map, you can add contiguous RAM at least to A0000H (655K). The sense switches are only examined by the initialization software when the machine is turned on, and the result is stored into a dedicated memory location that is referenced by the operating system thereafter.

Rick Wilton has donated a short utility program called MEMSIZE that can be added to your AUTOEXEC.BAT file to physically size RAM and reset the operating system's dedicated variable to reflect the true amount of memory. When MEMSIZE is invoked, it first checks to see if the sense switches were set for less than 544 Kbytes – if so, it simply exits. This is to protect users who have large amounts of memory but are using it for an "electronic disk" instead of making it directly available to applications programs.

If the sense switches are indeed set for the maximum, MEMSIZE attempts a write/read cycle to memory on each paragraph boundary up to the address A0000H, which is defined as the beginning of "reserved memory" by the IBM Technical Manual. It shifts the highest segment address off to the right to find the amount of Kbytes, then stores that value into the memory size variable. MEMSIZE is provided as Listing 3 (page 99), which is in a form compatible with the Microsoft Macro Assembler. ▶▶J

16-Bit Toolbox

Listing One (Text begins on page 92)

```
*************************************************************************
*                                                                       *
*       Convert 32 bit signed binary # to ascii string                  *
*       3/19/83 Steve Passe                                             *
*                                                                       *
*       Called values:          a1 == pointer to 8 byte buffer to       *
*                                       hold string.                    *
*                               d1 == 32 bit signed number.             *
*                                                                       *
*       Returned values:        d1 == (same as call value)              *
*                               d2 == digits in string.                 *
*                               a1 == address of first char in string.  *
*                                                                       *
*************************************************************************

                org     $880

binary_to_ascii
                movem.l d1/d3/d4,-(sp)
                moveq   #11-1,d2                buffer length minus dbf
init_buf        move.b  #' ',(a1)+              space into buffer
                dbf     d2,init_buf             more...
                moveq   #0,d2                   clear char count
                tst.l   d1                      is it
                bpl.s   div_loop                 neg?
                neg.l   d1                      make posative for now
div_loop        divu    #10,d1                  strip digit from right
*
                bvs.s   overflow                overflow flag set?
                move.l  d1,d3                   d3 -> remainder/xxx
                and.l   #$0000ffff,d1           erase remainder
                bra.s   make_ascii              skip overflow stuff
overflow
                move.w  d1,d3                   prepare for double div
                clr.w   d1                      zero low word
                swap    d1                      high word into low
                divu    #10,d1                  divide high word
                move.w  d1,d4                   save quotient
                move.w  d3,d1                   low word into low
                divu    #10,d1                  divide low word
                move.l  d1,d3                   d3 -> remainder
                swap    d1                      R/Q -> Q/R
                move.w  d4,d1                   d1 -> L/H
                swap    d1                      d1 -> H/L
*
make_ascii      swap    d3                      remainder into low word
                add.b   #'0',d3                 make ascii
```

(Continued on page 96)

16-Bit Toolbox

Listing One (Listing continued, text begins on page 92)

```
                move.b  d3,-(a1)            fill buffer from right
                addq.b  #1,d2               inc count
                tst.l   d1                  gone zero yet?
                bne.s   div_loop            no...
                movem.l (sp)+,d1/d3/d4      restore call val, work reg.
                tst.l   d1                  was it negative?
                bpl.s   all_done             no
                move.b  #'-',-(a1)          add the '-'
                addq.b  #1,d2               inc count
all_done
                rts

                end
```

End Listing One

Listing Two

```
*************************************************************************
*                                                                       *
*       Convert ascii string to binary value                            *
*       3/1/83  Steve Passe                                             *
*                                                                       *
*       Called values:          a1 == pointer to ascii string rep. of   *
*                                     decimal number. Legal string      *
*                                     chars: +-.0123456789              *
*                                                                       *
*       Skips leading blanks, terminating on first illegal char.        *
*                                                                       *
*       Returned values:        d1 == signed binary result              *
*                               d2 == digits after decimal point,       *
*                               a1 == address of first illegal char.    *
*                                                                       *
*************************************************************************

                org     $800

ascii_to_binary
                movem.l d3-d4,-(sp)         save registers
                moveq   #0,d1               clear answer
                moveq   #-1,d2              no decimal to start
skip_ws         cmp.b   #' ',(a1)+          look for space...
                beq.s   skip_ws             skip on...
                move.b  -(a1),-(sp)         save (possible) sign
                cmp.b   #'+',(a1)           look for pos. sign
                beq.s   skip_sign           skip sign
                cmp.b   #'-',(a1)           look for neg. sign
                bne.s   convert             ready for first char
skip_sign       addq    #1,a1               inc string pointer
convert         move.b  (a1)+,d3            into data reg
                cmp.b   #'.',d3             look for decimal point
                bne.s   digit               nope
                moveq   #0,d2               clear count
                bra.s   convert             continue digit search
digit           cmp.b   #'0',d3             low end numeral
                bcs     scan_done           less than '0'
                cmp.b   #'9',d3             high end
                bhi     scan_done           greater than '9'
                and.l   #$0f,d3             make binary
```

(Continued on page 98)

Listing Two (Listing continued, text begins on page 92)

```
*
                move.l  d1,d4                   get high word...
                swap    d4                      ... into low word
                mulu    #10,d1                  previous count, low word, * 10
                mulu    #10,d4                  high word also
                swap    d1                      high word into low,
                add.w   d4,d1                   add high to d4/low
                swap    d1                      restore order
*
                add.l   d3,d1                   add new digit
                tst     d2                      past decimal?
                bmi.s   convert                 not yet

                addq.b  #1,d2                   yes, one more past
                bra.s   convert                 again
scan_done       subq.l  #1,a1                   align on first fail
                move.b  (sp)+,d3                get sign
                cmp.b   #'-',d3                 was it neg?
                bne.s   not_neg                 no...
                neg.l   d1                      yes...
not_neg         movem.l (sp)+,d3-d4             restore work registers

                rts

                end
```

End Listing Two

16-Bit Toolbox
Listing Three

```
                        name    memsize
                        page    55,132
                        title   'MEMSIZE -- reset PC/DOS memory size variable

                    ;
                    ; MEMSIZE -- reset PC/DOS memory size variable at location 40:13h
                    ; ( for IBM PC under PC-DOS 1.1 -- See Technical Reference p. A-2 )
                    ; Version 1     April 1983
                    ;
                    ; Richard Wilton
                    ; Laboratory Microsystems
                    ; 4147 Beethoven Street
                    ; Los Angeles, CA  90066
                    ;
                    ;

0000                cseg    segment byte
                        assume  cs:cseg,ds:cseg
0100                    org     100h

0100  BB 0040       start:  mov     bx,40h          ; set DS:BX to point ...
0103  8E DB                mov     ds,bx
0105  BB 0013               mov     bx,13h          ; ... to memory size variable
0108  81 3F 0220            cmp     word ptr [bx],220h ; is current mem size < 544K ?
010C  7C 28                 jl      exit1           ; exit if it is

010E  B8 8800               mov     ax,8800h        ; otherwise, set up test loop
0111  BB 0000               mov     bx,0
0114  3D A000       loop1:  cmp     ax,0a000h       ; is AX = beginning of "reserved" addrs ?
0117  74 0F                 je      exit2           ; yes, so exit
0119  8E D8                 mov     ds,ax           ; no, so use this as a segment
011B  89 07                 mov     [bx],ax         ; write contents of AX to DS:BX ...
011D  8B 0F                 mov     cx,[bx]         ; ... and read it back to CX
011F  3B C1                 cmp     ax,cx           ; does data read = data written ?
0121  75 05                 jne     exit2           ; exit if it doesn't
0123  8C D8                 mov     ax,ds           ; otherwise, copy DS to AX ...
0125  40                    inc     ax              ; ... increment it ...
0126  EB EC                 jmp     loop1           ; ... and loop

0128  B1 06         exit2:  mov     cl,6            ; number of bits to shift
012A  D3 E8                 shr     ax,cl           ; convert segment to Kbytes
012C  BB 0040               mov     bx,40h          ; again, set DS:BX ...
012F  8E DB                 mov     ds,bx           ; ... to point to memory size variable
0131  BB 0013               mov     bx,13h
0134  89 07                 mov     [bx],ax         ; store updated memory size
0136  CD 20         exit1:  int     20h             ; exit to DOS
0138                cseg    ends                    ; end of code segment
                        end     start           ; end of assembly -- start addr at "start"
```

End Listing Three

OF INTEREST

by Michael Wiesenberg

Cachet for a Chameleon

The Chameleon, by **Seequa**, an IBM PC-compatible dual processor (8088 and Z80A) portable computer, has been added to the General Services Administration's (GSA) "Authorized ADP Schedule for Microcomputers, Peripherals and Support Equipment." This is tantamount to pre-approval of purchases of a product because government agencies can buy products from the list as needed, without having to go through a bidding process. What Seequa is implying is that if the government thinks their computer is good, you should too. This 28-pound portable – small enough to fit under an airplane seat – costs $1995, and comes with two 5¼-inch 160K disk drives, 128K RAM (internally expandable to 256K, and 704K externally), and 8K ROM (internally expandable to 40K). Also included are a 9-inch green phosphor screen (80 by 25 characters), standard color capability with external monitor, serial and parallel ports, and fan. The "standard package" contains four pieces of 16-bit software: MSDOS, MBASIC, PerfectWriter, and PerfectCalc. They don't supply it, but CP/M-80 software can be run. **Reader Service No. 101.**

Yet Another Apple Clone

The **Icon II**, from **Icon Computer**, costs $849, is completely compatible with Apple II and Franklin Ace 1000, has 64K, a 15-key numeric pad, and a complete 90-day warranty, optionally increasable to one year. **Reader Service No. 103.**

Little IBM Talks to Bigger IBM

The **PC Express** from **Intelligent Technologies** transforms an IBM PC into a communications workstation *that talks to other* PCs, directly accesses several mainframes, handles electronic mail, and performs automated telephone management. The package is software and tutorials on diskette and a board to plug into the PC or PC-compatible computer. Several communications protocols are offered, including synchronous and asynchronous. The PC Express I communicates with standard asynchronous mainframes and other micros, and costs $895. The PC Express II uses SNA to provide 327X emulation and asynchronous protocols for DEC VT-100/52 terminal emulation, and costs $1295. Both systems have automatic answer, automatic dialing from a data base of phone numbers, detection of busy, ringing, and voice answer, a full text editor to create those deathless messages, a 300 baud chip, and an RS-232 connector for an external modem with settings up to 19.2 kilobaud. The electronic mail capabilities permit automatic document routing through Western Union, The Source, CompuServe, or any personal computer with auto-answer and asynchronous communications. Data can be sent in both directions simultaneously. A PC can be a host to another local or remote PC or ASCII terminal, so that programs can be invoked, run, and modified locally or remotely. **Reader Service No. 125.**

Incompatible Formats Talk to Each Other

Disk format incompatibility has long been a problem in transferring files from one computer to another. The **Transporter** from Workman and Associates claims to be the solution to the problem. Two copies are not required of the program; only the receiving machine must run CP/M; and the Transporter can send itself to the other machine to establish two-way communications. Supposedly any two CP/M machines can be talking to each other in 20 minutes, as long as they have compatible communications ports and speeds; or, failing that, modems can be used. The included manual explains, they say, "the process of communication between CP/M machines." It also discusses problems of file transfer, from baud rates to port assignments. A modem program is also on the disk, as are other file transfer utilities. $69.50 postpaid. **Reader Service No. 107.**

Intelligent Print Buffer

ANGEL from **Ligo Research** is an intelligent print buffer, with 64K RAM, serial and parallel input/output interfaces, serial baud rate from 110 to 19.2K, and externally switch-selectable parity and data bit size. Serial and parallel can be switched one to the other, which, they claim, eliminates incompatibility between computers and printers. "Page mode" permits skipping pages, reprinting pages that might have jammed in the printer; and "page pause" permits single sheet printing. These modes are software- or switch-selectable. Space compression extends the effective buffer size to 128K. $295. **Reader Service No. 104.**

Trends: Bargain Computers

I just bought a real computer for $40. My **Timex-Sinclair 1000 Personal Computer**, which had been retailing for $69.95, was on sale for $54.99 at Long's. Until April 30, Timex was giving a $15 rebate, so my Z80 computer cost $39.99 (plus tax). Yes, I bought it at a drugstore. No, Long's won't give me any support, but Timex has a good warranty. There are several users' groups. A magazine and a number of newsletters are devoted to the little machine. Not much memory, you say? Several companies, including Timex, offer a 16K expansion for under $50. **Memotech** offers a 64K RAM for $179.95. You don't like the membrane keyboard? Memotech also has an expansion keyboard that plugs directly into the back of the Timex for $99.95. Timex has a hundred-dollar printer. No wonder this is the best-selling computer in the world. Every company should have an under-$50 starter computer. When the beginners get interested in computing, then sell them the more expensive peripherals, or the beefed-up model. Or better yet, make the inexpensive model upwardly compatible with the more powerful versions. Atari? Apple? Radio Shack? You guys listening? Commodore and TI seem to be getting the idea. The VIC 20 and the 99/44A are dropping to the hundred-dollar range. **Reader Service No. 111.**

Smart Optical Mouse

OptoMouse 2000 and **4000**, from **USI**, are optical mice whose motions on a reflective grid are translated into input to a computer's RS-232 port. The 2000 costs $179 and interfaces directly with the IBM PC and BASIC, and has a WordStar overlay optionally available for $49.95. Microprocessor-controlled, with two buttons, OptoMouse interfaces with any software that has cursor positioning capabilities. The 4000 has four buttons, on-board 8051 microprocessor optical system and interface circuitry, eight selectable baud rates from 110 to 9600, and, presumably, costs more (they didn't list a price). **Reader Service No. 113.**

6809 Single Board Micro

The **6809 Control Module** from **Wintek** is a Motorola 6809-based, single-board computer for dedicated control, protocol conversion, instrumentation, communications, robotics, etc. The 4.5-by-6.5-inch card has a microprocessor, a timer, a real-time clock, two RS-232 serial ports, four parallel ports with handshaking, and can accommodate up to 64K of RAM and EPROM. The 6809 Control Module is compatible with all Wintek's I/O modules, including serial, parallel, IEEE-488, driver/sensor, floppy disk interface, analog interface, parallel breadboard, console I/O, cassette, and counter/timer modules. Wintek also sells developmental software for the 6809. The boards cost $245 each, with up to a 50% discount for quantity purchases. The manual costs $5. **Reader Service No. 115.**

Inexpensive Graphics Plotter

The **Strobe 200**, from **Strobe Graphics Systems**, holds many different kinds of widely available pens, comes with software to produce line and bar graphs, pie charts, and print alphanumerics. It plots applications software, prints overhead slides, and, on Apple only, plots VisiCalc DIF files and dumps the screen display directly. The 200 costs $795 for IBM PC, Apple, or CP/M 2.2. An on-board RS-232C serial interface has four built-in foreign language character sets, proportional spacing, and costs $195. Parallel interface cards are available for Apple and Apple-compatible computers ($85), Osborne ($90), and Commodore and TRS-80 ($110). The Enhanced Business Graphics Package, a spreadsheet graphics program, requiring 64K, costs $195. What Strobe calls the "industry standard" Strobe 100 costs $699. **Reader Service No. 117.**

Is "Comprehensive Relational Data Base Management" a Buzzword?

DBPACK-II, from **Compu-Draw**, is a comprehensive relational data base management for CP/M and CDOS. The number of files (relations) that can be opened simultaneously is limited only by available memory. Each relation can have up to 65,535 records, and each record up to 65,535 fields. Individual relations can span several disks. Fields can be any of several data types, including text string data of up to 254 characters, word- or line-oriented text of unlimited length (variable record length data type), floating point, scientific floating point, composite numeric data with up to three subfields, date data, and others. Relations are automatically opened as needed. Several relations on a disk can be searched until the one with the required field is found. Modes include interactive menu, interactive command, and batch command, with no limit on the nesting of command files. On-line help is available at all times. Sorts or any other relations can be performed on the basis of an unlimited number of keys. Data entry with user-specified error checking is easily done. There is password protection at the relational level. Reports are easily generated and printed in many formats. DBPACK-II's programming structures include DO, WHILE, and REPEAT loops, and CASE and IF THEN ELSE constructs. A user-specified log of all activity during a session can be sent to a disk file or printer. The complete package costs $395. **Reader Service No. 119.**

PC Becomes Piano

Turn your IBM PC into a playable musical instrument with **Visible Piano** from **Synthalytics**. You get a plastic keyboard overlay that makes the keys look and feel "much like a standard piano keyboard." The space bar acts like the sustain pedal on a piano. Without additional hardware, this software package uses the monochrome or color graphics display and built-in speaker to both play notes and display them in standard pitch notation. Scores can be printed, with a 200-note buffer for page-length printing. In edit mode, you can change a score with full cursor control. Visible Piano has an eight-octave range. For $45 you get disk, overlay, instructions, and a songbook. **Reader Service No. 121.**

LISP in 8, 16, and 32 Bitth

Cromemco is offering LISP on its 8-, 16-, and 32-bit **Systems One**, **Two**, and **Three**, with virtual storage, string processing, fixed- and floating-point arithmetic, error trapping, interfacing with non-LISP procedures, a library of over 150 utilities, macro facilities, and a table-driven, user-modifiable parser that can redefine the scanner and define new LISP syntax. $595. **Reader Service No. 123.**

Pearson Correlations on Your Micro

SL-MICRO, from **Questionnaire Service Company**, is a statistical language for microcomputers. This statistical package processes data files with up to 32,600 cases and 200 variables. A free-to-fixed field conversion utility makes SL-MICRO compatible with data base packages. It has variable and value labels, long variable names, missing values, English error messages, and offers these procedures: Frequencies (with statistics and histogram), Crosstabs (including chi-square), Pearson Correlation (with an optional covariance table), and Multiple Regression (with Blockwise and Stepwise inclusion levels). You need CP/M on Z80 with two disk drives and 48K, or CP/M-86 with 128K and CBASIC. $250 including manual, or manual alone for $15. **Reader Service No. 125.**

Hot Rumor

You may read it here first, particularly if you don't get *Unique* (formerly the UNIX Software List), the newsletter for UNIX and C users published by **InfoPro Systems**. "FLASH! You read it here first (as usual)!" says *Unique*. They then go on to predict that this year AT&T will be offering a personal computer, a DBMS for UNIX, and a MAC-32-based desktop UNIX machine. *Unique* is $54 for a year of monthly issues ($12 more for overseas airmail), but discounts are available for nonprofit organizations, schools, bulk

and multi-year subscriptions, and individuals at a home address. Each additional subscription sent in the same envelope is $10 per year extra ($15 overseas). **Reader Service No. 127.**

Most of What You Ever Wanted to Know About the VIC 20

Next to Timex, the VIC 20 is perhaps the best-selling computer in the world. The **VIC 20 User Guide**, by John Heilborn and Ran Talbott, from **Osborne/McGraw-Hill**, is addressed to all levels of expertise. The book has operating instructions for computer and peripherals, tutorials in Commodore BASIC (including instruction and demonstration programs for color graphics and sound), and detailed programming instruction and reference material. System architecture, memory usage, pinouts, conversion tables, trigonometric functions, and sound and display characters codes are all said to be thoroughly covered. $14.95. **Reader Service No. 129.**

Something for the Kiddies?

Above Intelligence, from **Eberhard Engineering, P.C.**, is "thought provoking and entertaining software, targeted for the sophisticated executive." It includes 25 games and brain teasers, "ideal," they say, "for children, or people without computer experience." The games include Hangman and a word puzzle, it draws a Snoopy calendar, and "there is something for everyone." But, wait a minute, they emphasize that the "diskette also includes 16 Nudes, and Nudes with Calendars." Hmmm. Above Intelligence runs on TRS-80 Models II, 12, and 16 with Daisy Wheel II or line printer, and costs $49.95. **Reader Service No. 131.**

Calling Forth

For any interested Forth folks, we recently learned of two upcoming fall conferences.

For beginning or professional Forth enthusiasts, the 5th Annual Forth Convention will be held this *October 14-15* at the Hyatt Palo Alto in Palo Alto, California. Focusing on the future of Forth-based systems, topics will include Forth-based instruments, operating systems, and machines at the hardware level. Anyone interested in submitting abstracts should keep them to less than 100 words, and get them in by August 1st. To get an author's kit, write to: The Forth Interest Group, Convention Program Chairman, P.O. Box 1105, San Carlos, CA 94070 (or call the FIG hot line at 415-962-8653).

This November 23-25, the 5th Annual Forth Modification Laboratory (FORML) Conference will be held at the Asilomar Conference Center in Pacific Grove, California. Not intended for beginners, the conference will focus on guiding the evolution of Forth. Suggested topics include: Hardware Forth Implementation, Large Address Space Environments, Multiprogramming Architectures, Nucleus Variations, Operating System Environments, System Generation Techniques, Program Development Methodologies, and Applications. Anyone interested in submitting an abstract should also keep it to no longer than 100 words, and have it in by September 6th. To obtain registration information and an author's kit, write to: FORML, P.O. Box 51351, Palo Alto, CA 94303.

DDJ

(Continued from page 10)

I have "Easter egg" hunted for that information for several months now. After reading Gene's article, the Digital Research instruction began to make sense. I still don't understand how they assign the DPB to the various drives. My CP/M is Lifeboat, written for North Star. I know North Star's method of turning "on" the bits for the different drives, but I can't seem to find the "proper byte" to do that bit. Anyway, I would like to be able to have my disk drives access different DPB's. I have a BASF DSDD, Tandon DSDD and two Shugart SSDD, so you see my dilemma when the Tandon tracks faster than the BASF, and the Shugart is pulling up the rear like the tortoise.

I sincerely enjoy your magazine, having been an intermittent subscriber since *DDJ* No. 28. Thank you for assistance and the rest and enjoyment that the *DDJ* brings me.

Very truly yours,
J. E. McMurray
P.O. Box 6772
Columbia, SC 29260

Solutions to SRA

Dear Sirs:

In the article "Shifts and Rotations on the Z80," on the SRA (Shift Right Arithmetic) instruction, the author says: "There may be some practical reason for leaving bit 1 on, but I am not aware of it."

The reason for this is simple. SRA is an *arithmetic* shift and therefore retains the sign of a 2's complement signed number. SRA's valid input range is -128 to +127 while SRL (which is unsigned) goes from 0 to 255. SRA is used on the MS byte and is then followed by RR instructions for the rest of the bytes for multi-byte precision signed numbers.

R. Tim Coslet
2001 Manhattan, No. 317
Palo Alto, CA 94303

More Small-C Fixes

Dear Doctor Dobb,

Users of the Small-C Compiler version 2 (*DDJ* Nos. 74 and 75) may be interested in the following modifications, which improve the code generated by the compiler:

(1) One of the "optimizations" in the function peephole() actually replaces good code with worse, trading the code sequence

```
POP D
MOV A,L
STAX D
```

for a subroutine call. Both sequences are three bytes long, but the original code executes faster. Lines 288-290 and 351-353 in peephole() should be deleted.

(2) The function loadargc() puts the number of arguments in a function call into register A, using the code MVI A,n. In the special case when there are no arguments, A can be cleared by the single-byte instruction XRA A. To have the compiler do this, change lines 48-50 of loadargc() to

```
if (val)
{ ot("MVI A,"); outdec(val); nl();}
else
    ol("XRA A");
```

(3) The following changes improve the code generated for an indirect function call. Changes are needed in three places First, add the following to the runtime library:

```
CCDCAL: ENTRY
        PCHL
```

and make the concomitant changes to listings four and five of the compiler. Second, change the code generator function callstk() to

```
callstk()
{ call("CCDCAL");
}
```

Third, change lines 209-217 of callfunction() to

```
while (streq(1ptr, ")" == 0)
{  if (endst()) break;
   if (ptr)  /* direct call */
   {  expression(&const, &val);
      push();
   }
   else   /* calling HL */
   {  push();
      expression(&const, &val);
      swapstk(); /* get call address
          back in HL */
   }
   nargs = nargs + BPW;
   if (match(",") == 0) break
}
```

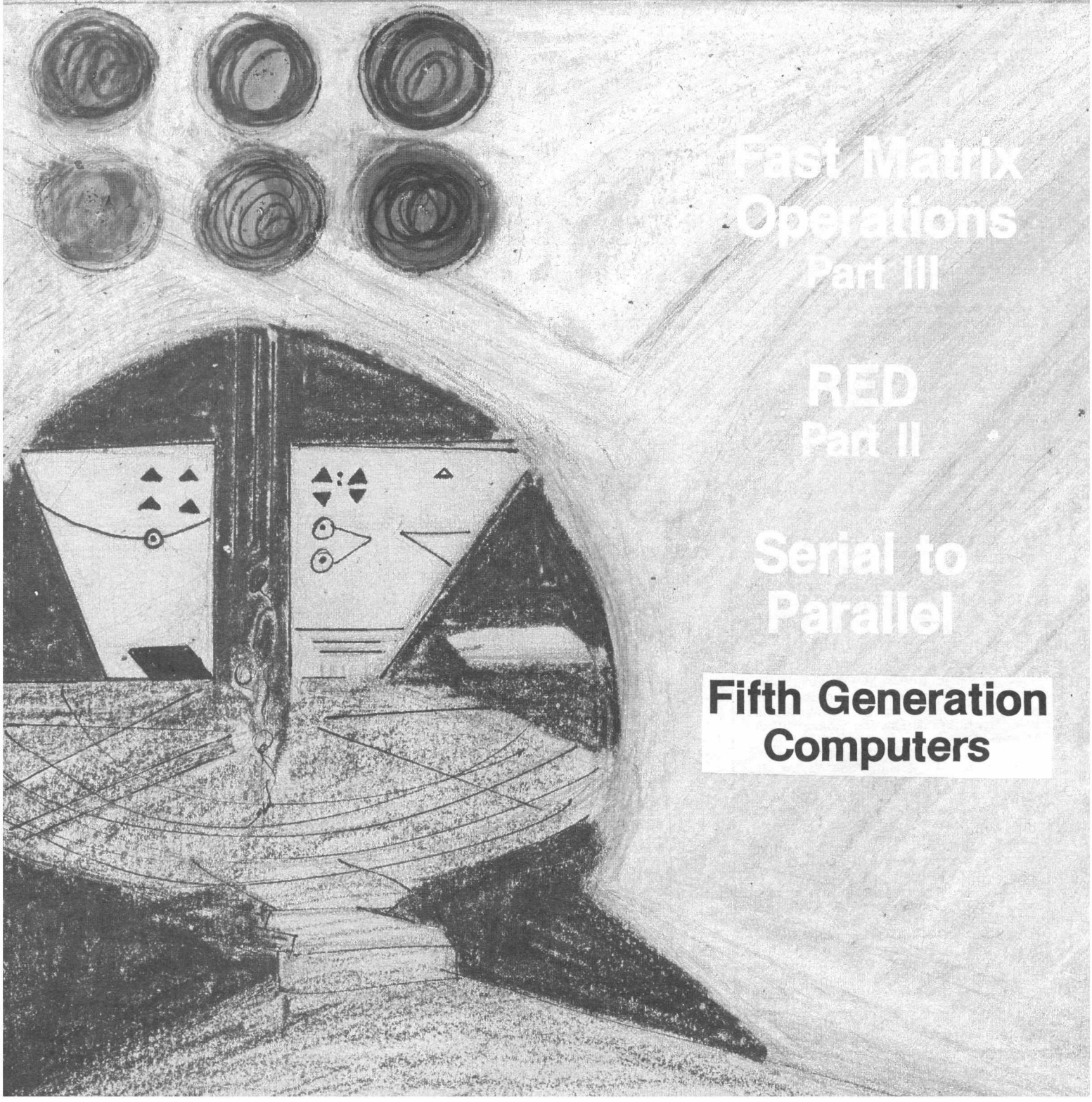
These changes save three bytes of code and 12 clock cycles (8080 timing) for each indirect function call, using the old 8080 programming trick of faking a dynamic call (i.e., a call to the address in HL) by a call to a subroutine which is just the instruction PCHL.

Sincerely,
Paul F. West
401 Nob Hill Pl., No. 5
Ann Arbor, MI 48103

#82 August 1983 $2.50 ($3.00 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems



Fast Matrix Operations Part III

RED Part II

Serial to Parallel

Fifth Generation Computers

Dr. Dobb's Journal
For Users of Small Computer Systems

August 1983 Volume 8, Issue 8

Publisher – Clifford West
Editor – Reynold Wiggins
Managing Editor – Craig LaGrow
Contributing Editors – Robert Blum, Dave Caulkins, Dave Cortesi, Ray Duncan, Michael Wiesenberg
Marketing Director – Beatrice Blatteis
Marketing Assistant – Sally Brenton
Advertising Director – Carl Landau
Advertising Sales – Doug Millison
Circulation Manager – Gloria Romanoff
Circulation Assistants – Terri Marty, Linda Marohn, Lew Griffin, Barbara Morgan-Nwuzi
Production Manager – Barbara Ruzgerian
Production Assistants – Alice Hinton, Shelley Rae, Marianne Tryens
Typesetter – Paula Fairchild
Cover Illustration – Kim Alderman

Copyright © 1983 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: DeWitt S. Brown, Burks A. Smith, Robert M. Connors, Robert C. Luckey, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Thomas Davis, Jan Steinman, G. Hunter, Ronald E. Johnson, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III, Infoworld, Stan Veit, Western Material Control, S.P. Kennedy, *Hatch*, Richard Jorgensen, John Boak. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

CONTENTS

ARTICLES

28 Serial-to-Parallel – A Flexible Utility Box
by Alan D. Wilcox
Limited to an RS-232C serial port, the author describes how he built a converter with parallel output, strobe, and handshaking. Included are a schematic and parts list.

36 McWORDER – A Tiny Text Editor
by William A. McWorter
The author finds the use and improvement of this little text editor as much fun as its creation. Written in BASIC, the 54-line program should provide an interesting project for those who enjoy making modifications and extensions.

44 Fast Matrix Operations in Forth, Part III
by Steven A. Ruzinsky
In this final installment of the fast matrix operations series, the author demonstrates some applications for the utilities that he defined in his earlier articles. Included are examples of image processing and least squares regression.

54 And Still More Fifth Generation Computers!
by Richard Grigonis
In the December 1982 issue, this author outlined his vision of computers in the 1990s. He is back this month with a more detailed discussion of how he thinks future micros will evolve, as well as a description of a fifth-generation workstation.

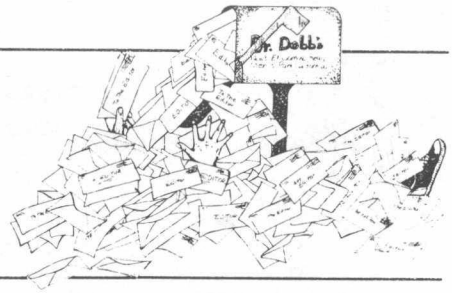
62 RED – A Better C Screen Editor, Part II
by Edward K. Ream
Last month, we got a look at RED's buffer routines. This month the rest of the Small-C screen editor is presented, including the new Small-C I/O library for the buffer routines.

DEPARTMENTS

6 Letters
6 Editorial
10 Dr. Dobb's Clinic
Specialist Symbols and I/O Benchmarks for CP/M Plus
74 CP/M Exchange
CP/M Plus Memory Management; Zero Length File Test; and PAUSEIF, QUITIF, and Now SKIPIF
108 Software Review
ACTxx Cross Assemblers
111 Book Reviews
Microcomputer Disk Techniques; Microcomputer Design and Construction; Pascal Implementation: The P4 Compiler; Projects in Machine Intelligence for Your Home Computer; A Practical Introduction to Computer Graphics
118 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

LETTERS



Mending, but Commending

To the Editor:

An error appeared in two of the figures illustrating "B-Tree ISAM Concepts" in your June issue. In Figure 3, the root node should show "L" in place of "I." Similarly, both root nodes in Figure 4 should show "R" instead of "K." Unless I've missed something, the network shown in Figure 3 could never find key "K" because the comparison at the root node would send it down the wrong path.

Otherwise the article was excellent, as was your June issue as a whole. The article raised a question in my mind. There's a design trade-off between a many-layered tall tree and a flatter tree with more elements at each node, and therefore more reliance on sequential processing within each level. What are the considerations that would affect the choice of "height" versus "thickness" in a particular application?

Please continue to feature the IBM Personal Computer in your articles. I intend to use both programs that you ran this month in my study of assembly language. I also find the articles on computer science and applications in higher-level languages very helpful. I even skim through the articles on CP/M to pick up some useful concepts. Please keep up the good work.

Sincerely,
Michael G. McCarter
Reliance Ins. Co.
4 Penn Center Plaza
Philadelphia, PA 19103

A "Fix" for the IBM Graphics Printer

Dear Editor,

Thanks for publishing "Printing Graphics on the IBM PC," in your June 1983 issue. As I mentioned in the article, the differences in control ROMs for the Epson printer can cause trouble. I now believe that users attempting to implement the program I submitted, on the IBM Graphics Printer, will have some minor problems. Basically, the control ROM for the IBM Graphics Printer is different from that available from Epson under their GRAFTRAX option.

The "fix" for users equipped with the IBM Graphics Printer is simple. The 20th line of the listing reads as follows:

```
LSPC  DB   27,'A',8,13,10,'$'
           ;Sets printer to 8/72"
```

Changing the line to:

```
LSPC  DB   27,'A',8,27,'2',13,10,'$'
           ;Sets printer . . . .
```

should correct the problem. I still have not been able to locate a configuration equipped with the Graphics Printer, so I haven't verifited the fix. The data I do have indicate that if there is a problem, this should correct it. Basically, the Epson with GRAFTRAX accepts the Escape 'A' sequence and acts on it immediately. The IBM version sets line spacing on the Escape 'A' sequence, but does not implement the new spacing until it receives an

EDITORIAL

Once again, we've published our largest issue ever! Our expansion is exciting; but at the same time, we are conscious that bigger is not always better. In this case, however, our increase in size will allow us options we haven't had in the past. With more editorial pages, we have the opportunity to present more variety each month, even when we run larger pieces, as we have in this issue and in the last.

Among the things we've wanted to devote more space to are reviews of books, software, and occasionally hardware. We have recently been updating our lists of reviewers, and feel that this is an appropriate time to extend an invitation to potential reviewers. If you are interested in doing reviews for *Dr. Dobb's* – whether on books, software, or hardware – drop us a note outlining your qualifications and interests. We'll get right back with you concerning particulars.

* * *

If you have ever wanted to have an impact on the quality of articles in our magazine, this could be your big chance. Since *Dr. Dobb's* is, among other things, a reader forum, our readership has traditionally filled gaps and corrected any occasional oversights. Even so, this feedback mechanism can be slow and cumbersome, and getting comments into the Letters column can take a while. Recent mail pointing out some of our blind spots set us to thinking. As the microcomputer industry has become increasingly complex and diverse, it has become difficult for any of us to stay on top of every area. Our current technical advisors have done an excellent job; but as the variety, specificity, and volume of material for the magazine has increased, so has the need for greater breadth in our resources. Consequently, we think it appropriate to include among our advisors a board of referees. The function of the board will be to provide insights on specific technical topics, and to make suggestions regarding material for *DDJ*. Since much of our best input comes from our readers, we can think of no better place to turn for qualified people.

We're looking for experts in various subject areas to review perhaps two to four manuscripts per month. The subject categories include, but are not limited to: mathematics, graphics, programming languages (of all sorts), utilities, and operating systems (including MS-DOS, CP/M-86, CP/M-68K, UNIX and UNIX look-alikes). This addition to our technical crew should ensure more accurate and complete information, and provide a better level from which to begin the usual reader dialogue.

In gratitude for their generous service, referees will receive a complimentary subscription to *DDJ*, and will have their names and professional affiliations listed in the magazine. Those who are interested in participating should contact us with information about their areas of specialty and interest. We'd love to talk with you.

Reynold Wiggins

Escape '2' sequence. On the GRAFTRAX, incidentally, this latter sequence resets the printer to "power up" spacing.

I hope this has not caused too many of your readers a problem.

I haven't been doing much, but I'll be back in touch one of these days. I'm waiting for the CI C compiler right now, so maybe I'll be able to submit some C in the future. That might reduce the listing volume to something less than encyclopedic dimensions.

Dan W. Daetwyler
Route 5, Box 518A
Springdale, AR 72764

C Clearly With Descriptive Operators

Dear Editor:

The C systems programming language is powerful and concise – sometimes too concise. Indistinct operator names cause confusion and error. These three pairs of operator names cause the greatest trouble:

= (assignment) vs. == (equality test)
& (bitwise and) vs. && (logical and)
| (bitwise or) vs. || (logical or)

The subtle differences in the meanings of the bitwise and logical operators aggravate the problem. Fortunately, a remedy exists.

Making operator names more distinct reduces confusion. The C preprocessor offers a simple way to do this. For example:

```
# define   and     &&
    /* logical and now written 'and'    */
# define   or      ||
    /* logical or now written 'or'      */
# define   equals  ==
    /* equality test now written 'equals'*/
integer x;
if (x equals 10)
    /* This replaces "if (x == 10)" and
       reduces errors like the use of
       "if (x = 10)."                   */
```

Using distinct operator names makes programs easier to read, write, and debug, making better use of the C language.

Sincerely,
Bryan M. Willman
309 Iowa Avenue, No. 3
Iowa City, IA 52240

A Reader's Rebuttal

Good morning,

I am writing you as a concerned scientist and subscriber. I am surprised that H.T. Gordon's article did not undergo as much scrutiny as many of your program listings do. As a mathematician and computer scientist I find his last paragraph disturbing, so let me explain and state a rebuttal.

- Every number theory book contains a chapter on divisibility, and therein are most, if not all, the "algorithms" described by Gordon.
- "Algorithms" are also found in many reference texts in computer science, such as Knuth's, *The Art of Computer Programming, Volume 1,* Addison-Wesley.
- These divisibility "algorithms" developed by the author are basically related to the mod function (X is divisible by 9 if and only if x mod 10 is). This easily explains generalizations to other bases.
- Discovery of these algorithms dates back to the Chinese some 5000 years ago.
- Archimedes and Newton would have been offended at Gordon's statement about "computational drudgery, in the pre-computer era."
- There are algorithms for divisibility by any prime number.
- Gordon, in using the term "randomness," is referring to uniformly distributed random numbers. His 1/3 test does not work on the set {1,2,3, . . . , N} for N large.

I tried not to be caustic, but may have failed. Mathematics is an old science, and as such it has many followers. Please watch thy written words . . . .

Sincerely,
Dr. Georges Grinstein
Professor, Computer Sciences
Fitchburg State College
Fitchburg, MA 01420

IBM's Fortran Compiler – Where to Put the Blame

Dr. Dobb:

In the Letters section of the May issue Messrs. Glass and Landis presented some benchmark comparisons which include execution time of IBM's Fortran compiler written by Microsoft and implemented on the PC.

Lamenting the agonizingly slow execution time of Microsoft's Fortran, they lay the blame on Microsoft's Pascal, since the Fortran compiler is written in Pascal.

However, the execution of the target language (Intel 8086 machine language) should not be affected by whether the compiler is written in Pascal, BASIC, or whatever. After all, the compiler simply translates ASCII source code into the appropriate binary code.

Also, Microsoft Pascal is written in Microsoft Pascal, and it does very well in benchmark tests such as the algorithm in Jim Gilbreath's article, "A High-Level Language Benchmark," in the September 1981 issue of *Byte* magazine.

Perhaps the poor performance reported was in part caused by not turning off the default DEBUG faility, which can add considerably to execution time. If not, the blame is on the Fortran compiler – not the language it is written in.

Sincerely,
Daniel L. Lee
SourceWare
1401 E. 55th St., Suite 601
Chicago, IL 60615

DDJ

"The computer at work broke down today, and we all had to learn to think for ourselves."

DR. DOBB'S CLINIC

by D. E. Cortesi, Resident Intern

Specialist Symbols

Back in March we published a note from J. Dempsey, asking for solutions to the problem of editing, printing, and processing the kind of non-ASCII symbols that turn up in such professional specialties as Linguistics. Several readers responded, usually with suggestions to try one or another software product or with accounts of how they looked for solutions to the same problem with little more success than Dempsey had had. We passed copies of all those letters on to Dempsey; perhaps we will hear more from him later.

There seems to be general agreement that it isn't difficult to display specialist symbols on the screens of machines that support soft character sets (Atari, IBM PC, etc.). It may be quite difficult to get one's word processing software to handle them. There is no easy solution to the problem of keyboard entry, at least with existing products. The IBM PC's trick of entering three decimal digits through the numeric keypad with the Alt key held down is the most flexible one we know of from an OEM, but the best solution is for the software to provide the ability to map any key to any symbol. The PE editor for the PC has that feature (see last month under "Cheap Thrills"), but it's best when the ability is provided as part of, or as an extension of, the operating system.

There does seem to be a satisfactory way of printing specialist symbols. For evidence, see the letter from Prof. Dr. Neuhaus of Munster, reproduced on page 12. His letter is the best endorsement of the Fancy Font product that we've seen; in the original, it is clear that every character was produced by it, including the elegant letterhead.

We don't know how one represents a specialist symbol as input to Fancy Font. *If* it is by way of a single, non-ASCII byte . . . and *if* one could make one's screen hardware use the same byte value to display the same user-defined symbol . . . and *if* one's word processor could handle those byte values . . . and *if* one had an acceptable way of entering them from the keyboard . . . *then* one would have a pretty good start at handling the problem.

CP/M 3 Earns Its Plus

In June we went on at some length about how the sector-buffering scheme of CP/M Plus didn't seem to have much effect on the kind of sequential I/O done by the bulk of CP/M commands. This month we can report a much happier conclusion. We have made some tests of direct-access disk I/O under CP/M Plus, and it is very quick indeed.

Comparative benchmarks are not simple things to organize. It takes some thought to ensure that the comparative results are indeed comparable. We spent some time devising a set of BASIC programs that would exercise direct-access I/O in a realistic way, and yet which would be simple enough for other people to duplicate on almost any hardware. Take the time now to read through the programs.

The Test Programs

The first is MAKEDIR (Listing 1, page 18). It creates a direct-access file; the variables NUMBYTES and NUMREC determine its size. For a realistic test, the file must be large enough to span several extents (use several directory entries) and many tracks. Line 1040 must be adjusted to reflect the size of a physical disk sector, and line 1060 to reflect the size of an extent, in the test system. The same lines 1010 through 1070 appear in all the other programs and must be identical in all.

MAKEDIR writes a file sequentially, but it does it with direct-access I/O services. Its execution time will be dominated by I/O and should be about the same as if the file were being written with sequential operations.

In order to get comparable results, the test file should be written only once, and then used in place in all subsequent tests in both systems. If the file is erased and rewritten, the distance between data and directory and the span of tracks across the data may change, and that would make the results incompatible.

The second program is RANDINPT (Listing 2, page 21). It is a naive test of direct access: it simply generates 200 random integers in the range of 1 to NUMREC and reads the corresponding records. Its accesses will exhibit no clustering (or to put it another way, its "working set" is the whole file). Applications that process records in batches (and so are dominated by I/O time) will usually have some kind of ordering or clustering to their accesses. Interactive programs may retrieve randomly scattered records as RANDINPT does, but will not do so in a tight loop. The execution time of an interactive program will be dominated by keying delays. However, if we measure not the total running time of RANDINPT but its running time divided by the number of records it retrieves, we will have an idea of the speed at which *one* record can be retrieved. The time to retrieve one record may be an important factor in the response time of an interactive program.

Take special note of line 1005 in RANDINPT. It is essential to set the pseudo-random generator to a known value before every run. Otherwise we can't be positive that the results under the two systems are truly comparable.

When sectors are larger than records, I/O time is not determined by the number of logical records that are read. Two things are important: first, the number of disk sectors that must be transferred, and second, the number of times the system must go to the directory. RANDINPT counts these things in two different ways. Whenever the current record falls in a different sector from the prior one, it counts a "sector transition." In a system with a single sector buffer, every sector transition requires another sector to be read. Whenever the current record falls in a different extent than the prior one, RANDINPT counts an "extent transition." In a system without directory buffering, an extent transition forces a read of the directory. The program also keeps track of the number of unique sectors and extents that are accessed, in order to give an idea of how well the file was covered.

The third program is RANDUPDT (Listing 3, page 21). It visits the same sequence of records as does RANDINPT, but it also updates each record, forcing the system to write as well as read. Its mean time to update one record is again an indication of the response time of an interactive system.

The RAND programs are not representative of any real applications (except interactive ones, as noted). With the fourth program, SKIPUPDT (Listing 4, page 22), we attempt to model the class of applications that do batched updates of a direct-access database. Imagine an order-entry application in which incoming orders are entered interactively during the day. The line items of the orders are batched in a sequential file. Each evening, these transactions are sorted by part number, and then applied in sequence to update an inventory database. The resulting sequence of accesses is ordered in that it updates records in ascending order by key, but random in that not all records are

Prof. Dr. H. J. Neuhaus
Direktor des Englischen Seminars

Westfälische
Wilhelms-Universität
Johannisstraße 12 - 20
D-4400 Münster

West Germany

Mr. D. E. Cortesi
Dr. Dobb's Journal
Box E
1263 El Camino Real
Menlo Park, CA 94025
U. S. A.

April 6, 1983

Re: Dr. Dobb's Clinic; Number 77, March 1963 "Demsey's Dilemma"

Dear Mr. Cortesi,

Linguists have for some time been used to laser printers on main-frame computers that allow individual character definition and the use of special alphabets. Donald Knuth's **METAFONT** alphabet design system is one of the more powerful software tools available for that purpose.

On microcomputers there is a very professionly done package by SoftCraft, 8726 S. Sepulaveda Boulevard, Suite 1641, Los Angeles. The **Fancy Font** system was introduced in 1982. It has been used extensively in my department. We created various fonts. One of them is an implementation of the **International Phonetic Alphabet**. Line 3 of Mr. Dempsey's text example looks like this:

[dɔkkun darfʃun tʃi: bu i:n ʃi:ri:nforu:ʃ a:he vɛ]

There are special symbols for diphthongs: a͡i a͡u ɔy , nasals: ɶ̃ ɛ̃ , etc.

A major problem with special alphabets is their QWERTY keyboard representation and their logical ASCII value ordering. These are normally quite different. If Mr. Dempsey is interested he may contact me for further information.

Sincerely yours,

Figure 1.

updated. It is this kind of skip-sequential processing that SKIPUPDT attempts to model. It generates approximately 200 keys, each key being incremented by at least 1 but no more than 10 from the prior key.

Skip-sequential processing causes many fewer sector and extent transitions than does random processing. Accordingly we can expect that SKIPUPDT will have a much shorter running time than RANDUPDT, at least under CP/M 2.2 where the transitions have the greatest impact. Its total time should be indicative of the relative speeds of a real skip-sequential program in the two systems.

The last program, TREEUPDT (Listing 5, page 24), is an attempt to model the access patterns of a B+Tree database manager (see the tutorial by Deppe and Bartholomew in the June issue of *DDJ*). A tree-structured file has an heirarchical index. The topmost level is a single record of keys, which points to several records of keys, which point to many records of keys, which in turn point to lots and lots of data records.

The TREEUPDT program simulates a number of random updates of records stored in such a tree. It assumes that the topmost index record would be kept in storage, and that the next two levels are stored in adjacent records at the front of the file. It reads one of the first eight records to simulate retrieval of an index node; reads one of the next 64 records to simulate index retrieval at the next level; and finally updates one of the remaining records to simulate access to a data record. The per-transaction time of TREEUPDT should tell something about the performance of a tree-structured database.

The Results

These programs were run under CP/M 2.2 and CP/M Plus in the same system. The system is a moderately fast one as 8-bit systems go. Its CPU is a Z80 running at 4MHz; it has single-sided, double-density, 8-inch disk drives (Shugart 801s, 6 millisecond step rate). The BIOS for CP/M 2.2 uses separate sector buffers for reading and writing. The BIOS for CP/M Plus supplies 70Kb of sector buffers, as described in this column in June. The exact timings given below would be repeatable only in another system with exactly the same hardware and software. However, the *ratio* between the times for the two systems should be constant in any one system, with the single exception that the CP/M Plus times are dependent on the number of sector buffers available.

Under CP/M 2.2 the programs were loaded and run one after another under the MBASIC 5.0 interpreter. Under CP/M Plus the programs were run as separate commands, with a return to the operating system between each. That was necessary because CP/M Plus discards its buffered sectors each time the CCP takes control. If we hadn't exited from the MBASIC environment between each program, sectors brought in by one program would have still been available when the next one started.

Times were taken with a stopwatch and recorded to the nearest second. The results appear in Table 1 (page 16). Each row of the table represents a run. The column "No. of records" is the count of data records read or updated in the run. The "Total" columns show run time in seconds; the "Per rec." columns show run time divided by data records processed. The last column shows the ratio of CP/M Plus run time divided by CP/M 2.2 run time, as a percentage.

The first surprise appears in the first line: MAKEDIR ran slower under CP/M Plus. The difference in times is not very significant, but it is puzzling. CP/M Plus is usually about 10 percent *faster* at sequential I/O.

The lines headed RANDINPT (1024) and RANDUPDT (1024) show that CP/M Plus can be strikingly faster at pure random access. It read a random record in about one-fourth, and updated one in about one-fifth, the time taken by CP/M 2.2. One noticeable difference between the systems was that, under CP/M Plus, it took as long as 13 seconds to close a file. RANDUPDT caused many modified sectors to accumulate in storage. When the CLOSE statement was executed, the system spent many seconds dumping them all to disk. You would be well advised to issue frequent closes to "checkpoint" your random files under CP/M Plus.

The RANDUPDT test revealed a disadvantage of our two-buffered BIOS for 2.2; a BIOS with a single sector buffer might actually run faster on this test. The cause lies in this typical sequence of operations:

read data sector A
write to data sector A
read the directory
read data sector B
write to data sector B

The usual 2.2 BIOS would have to dump the updated sector A to disk before reading the directory. Ours didn't; it held sector A in the write buffer and read the directory sector(s) and sector B into its read buffer. Not until the call to write sector B did it need to purge its write buffer. Alas, by then it had moved the disk arm to the track of sector B, and had to do an additional seek before it could write sector A.

Under CP/M 2.2, the disk drive sounded a rhythmic "doo-wacka-doo" as it moved between the directory and the data area. Extent transitions were obviously dominating the run time of the RAND tests. In order to find out how much effect these had, we modified line 1030 of the two RAND programs to show only 128 records in the file. As a result, the programs confined their activity to the first extent of the file, eliminating extent transitions. The result of these runs is shown in Table 1 in the lines headed RANDINPT (128) and RANDUPDT (128). Compare them to the lines just above them.

It appears that extent transitions (i.e., directory accesses) account for about three-fourths of the per-record time under CP/M 2.2, but for less than

Table I

Summary of the results of direct-access I/O tests run on CP/M 2.2 and CP/M Plus.

Program	No. of Records	CP/M 2.2 Total	CP/M 2.2 Per rec.	CP/M 3.0 Total	CP/M 3.0 Per rec.	CPM3/CPM2 (as pct)
MAKEDIR	1024	48	0.05	53	0.05	110%
RANDINPT (1024)	200	164	0.82	43	0.21	26%
RANDINPT (128)	200	45	0.23	23[1]	0.12	51%
RANDUPDT (1024)	200	333	1.67	65[2]	0.33	20%
RANDUPDT (128)	200	73	0.37	32[1]	0.16	44%
SKIPUPDT	192	66	0.34	64[3]	0.33	97%
TREEUPDT	100	261	2.61	36[3]	0.36	14%

Note 1: this run was dominated by CPU time; the disk was idle during much of the run.
Note 2: includes file-close time of about 13 seconds.
Note 3: includes file-close time of about 6 seconds.

half of it under CP/M Plus. This demonstrates the effectiveness of the directory hashing tables and directory buffers used by CP/M Plus. Note that the CP/M Plus time for these runs was not dominated by I/O time; after the first 16 records or so, all data were in storage and the disk simply stopped moving.

The performance of the two systems was nearly identical on the SKIPUPDT test. Sector buffering has little effect when the program never returns to the same sector! A skip-sequential application, like a purely sequential one, will not run significantly faster under CP/M Plus. The tiny improvement that does show is probably due to directory hashing.

However, it seems that access to a tree-structured database will be greatly improved. The relative speed of CP/M Plus in the TREEUPDT test is really impressive: it ran seven times faster than CP/M 2.2! Indeed, under CP/M Plus the per-transaction time of TREEUPDT was almost identical to that of RANDUPDT. TREEUPDT makes two more disk accesses per transaction, but since these are accesses to the frequently used "index" sectors, they were almost always storage-to-storage moves rather than I/O operations. A database system that uses a B+Tree organization should show markedly better performance under CP/M Plus (would some reader care to verify that with a real program?).

Summary

In the area of disk I/O performance, then, it appears that CP/M Plus is not significantly slower than CP/M 2.2 for sequential and skip-sequential I/O, while its direct access operations, given a sufficient buffer area, are very much faster.

In June we said that a single 48Kb system bank would be sufficient extra hardware for CP/M Plus. That remains our recommendation for systems where direct access I/O is not a significant part of daily operations. When direct access is used a lot, or for important applications, we recommend that a CP/M Plus system be given one additional 48Kb bank for sector buffers (two, if the working set of the application is large). The improved performance should justify the cost.

A response on behalf of Digital Research to the June Clinic arrived too late to include in this issue. It will be presented in the Letters column next month. –Ed.

DDJ

(Listings 1-5 on pages 18-26)

Dr. Dobb's Clinic (Text begins on page 10)

Listing One

```
1000 ' MAKEFILE.BAS: create large direct-access file
1010 DEFINT A-Z
1020 NUMBYTES = 128      :' bytes per record
1030 NUMREC = 1024       :' records in file (128*1024=128KB)
1040 RECSEC = 1024\NUMBYTES :' 1K sectors in use
1050 NUMSEC = NUMREC\RECSEC :' disk sectors in file
1060 RECEXT = 16384\NUMBYTES:' 16KB extents in use
1070 NUMEXT = NUMREC\RECEXT :' directory extents in file
2000 PRINT "Opening file";CHR$(7) :' start the watch
2010 OPEN "R",#1,"TEST.DIR",NUMBYTES
2020 FIELD #1, 2 AS KEY$, 2 AS DAT$, NUMBYTES-4 AS FILL$
3000 FOR KEY = 1 TO NUMREC
3010    LSET KEY$ = MKI$(KEY)
3020    LSET DAT$ = MKI$(0)
3030    LSET FILL$ = STRING$(124," ")
3040    PUT #1,KEY
3050    PRINT KEY
3060 NEXT KEY
4000 PRINT : PRINT "Closing file"
4010 CLOSE #1
4020 PRINT CHR$(7);"File closed" :' stop the watch
9000 END
```

End Listing One

Dr. Dobb's Clinic (Listing continued, text begins on page 10)

Listing Two

```
1000 ' RANDINPT.BAS: retrieve many records, strictly at random
1005 RANDOMIZE(4093)     :' ensure repeatability
1010 DEFINT A-Z
1020 NUMBYTES = 128      :' bytes per record
1030 NUMREC = 1024       :' records in file (128*1024=128KB)
1040 RECSEC = 1024\NUMBYTES :' 1K sectors in use
1050 NUMSEC = NUMREC\RECSEC :' disk sectors in file
1060 RECEXT = 16384\NUMBYTES:' 16KB extents in use
1070 NUMEXT = NUMREC\RECEXT :' directory extents in file
1075 NEWSEC = 0 : OLDSEC = 0 : SECTRANS = 0
```

```
1080 NEWEXT = 0 : OLDEXT = 0 : EXTTRANS = 0
1085 DIM SECTOR(NUMSEC), EXTENT(NUMEXT)
2000 PRINT "Opening file";CHR$(7) :' start the watch
2010 OPEN "R",#1,"TEST.DIR",NUMBYTES
2020 FIELD #1, 2 AS KEY$, 2 AS DAT$, NUMBYTES-4 AS FILL$
3000 FOR COUNT = 1 TO 200
3010    KEY = INT(RND*NUMREC) : KEY = KEY-(KEY=0)
3020    NEWSEC = 1 + KEY\RECSEC
3030    SECTOR(NEWSEC) = 1
3040    SECTRANS = SECTRANS-(OLDSEC<>NEWSEC)
3050    OLDSEC = NEWSEC
3060    NEWEXT = 1 + KEY\RECEXT
3070    EXTENT(NEWEXT) = 1
3080    EXTTRANS = EXTTRANS-(OLDEXT<>NEWEXT)
3090    OLDEXT = NEWEXT
3100    GET #1,KEY
3110    PRINT COUNT, KEY, NEWSEC, NEWEXT
3120 NEXT COUNT
4000 PRINT : PRINT "Closing file"
4010 CLOSE #1
4020 PRINT CHR$(7);"File closed" :' stop the watch
5000 COUNT = 0
5010 FOR J = 1 TO NUMSEC
5020    COUNT = COUNT + SECTOR(J)
5030 NEXT J
5040 PRINT COUNT;" unique sectors visited of";NUMSEC
5050 COUNT = 0
5060 FOR J = 1 TO NUMEXT
5070    COUNT = COUNT + EXTENT(J)
5080 NEXT J
5090 PRINT COUNT;" unique extents visited of";NUMEXT
5100 PRINT SECTRANS;" sector transitions"
5110 PRINT EXTTRANS;" extent transitions"
9000 END
```

End Listing Two

Listing Three

```
1000 ' RANDUPDT.BAS: update many records, strictly at random
1005 RANDOMIZE(4093)     :' ensure repeatability
1010 DEFINT A-Z
1020 NUMBYTES = 128      :' bytes per record
1030 NUMREC = 1024       :' records in file (128*1024=128KB)
1040 RECSEC = 1024\NUMBYTES :' 1K sectors in use
1050 NUMSEC = NUMREC\RECSEC :' disk sectors in file
1060 RECEXT = 16384\NUMBYTES:' 16KB extents in use
1070 NUMEXT = NUMREC\RECEXT :' directory extents in file
1075 NEWSEC = 0 : OLDSEC = 0 : SECTRANS = 0
1080 NEWEXT = 0 : OLDEXT = 0 : EXTTRANS = 0
1085 DIM SECTOR(NUMSEC), EXTENT(NUMEXT)
2000 PRINT "Opening file";CHR$(7) :' start the watch
2010 OPEN "R",#1,"TEST.DIR",NUMBYTES
2020 FIELD #1, 2 AS KEY$, 2 AS DAT$, NUMBYTES-4 AS FILL$
3000 FOR COUNT = 1 TO 200
3010    KEY = INT(RND*NUMREC) : KEY = KEY-(KEY=0)
3020    NEWSEC = 1 + KEY\RECSEC
3030    SECTOR(NEWSEC) = 1
3040    SECTRANS = SECTRANS-(OLDSEC<>NEWSEC)
3050    OLDSEC = NEWSEC
```

(Continued on next page)

 (Listing continued, text begins on page 10)

Listing Three *(Continued)*

```
3060    NEWEXT = 1 + KEY\RECEXT
3070    EXTENT(NEWEXT) = 1
3080    EXTTRANS = EXTTRANS-(OLDEXT<>NEWEXT)
3090    OLDEXT = NEWEXT
3100    GET #1,KEY
3101    LSET DAT$=MKI$(1+CVI(DAT$))
3102    PUT #1,KEY
3110    PRINT COUNT, KEY, NEWSEC, NEWEXT, CVI(DAT$)
3120 NEXT COUNT
4000 PRINT : PRINT "Closing file"
4010 CLOSE #1
4020 PRINT CHR$(7);"File closed" :' stop the watch
5000 COUNT = 0
5010 FOR J = 1 TO NUMSEC
5020    COUNT = COUNT + SECTOR(J)
5030 NEXT J
5040 PRINT COUNT;" unique sectors visited of";NUMSEC
5050 COUNT = 0
5060 FOR J = 1 TO NUMEXT
5070    COUNT = COUNT + EXTENT(J)
5080 NEXT J
5090 PRINT COUNT;" unique extents visited of";NUMEXT
5100 PRINT SECTRANS;" sector transitions"
5110 PRINT EXTTRANS;" extent transitions"
9000 END
```

End Listing Three

Listing Four

```
1000 ' SKIPUPDT.BAS: update records in skip-sequential order
1005 RANDOMIZE(4093)    :' ensure repeatability
1010 DEFINT A-Z
1020 NUMBYTES = 128     :' bytes per record
1030 NUMREC = 1024      :' records in file (128*1024=128KB)
1040 RECSEC = 1024\NUMBYTES :' 1K sectors in use
1050 NUMSEC = NUMREC\RECSEC :' disk sectors in file
1060 RECEXT = 16384\NUMBYTES:' 16KB extents in use
1070 NUMEXT = NUMREC\RECEXT :' directory extents in file
1075 NEWSEC = 0 : OLDSEC = 0 : SECTRANS = 0
1080 NEWEXT = 0 : OLDEXT = 0 : EXTTRANS = 0
1085 DIM SECTOR(NUMSEC), EXTENT(NUMEXT)
2000 PRINT "Opening file";CHR$(7) :' start the watch
2010 OPEN"R",#1,"TEST.DIR",NUMBYTES
2020 FIELD #1, 2 AS KEY$, 2 AS DAT$, NUMBYTES-4 AS FILL$
3000 KEY = 0 : COUNT = 0
3003 WHILE KEY < NUMREC-10
3010    KEY = KEY + INT(RND*9) + 1
3015    COUNT = COUNT + 1
3020    NEWSEC = 1 + KEY\RECSEC
3030    SECTOR(NEWSEC) = 1
3040    SECTRANS = SECTRANS-(OLDSEC<>NEWSEC)
3050    OLDSEC = NEWSEC
3060    NEWEXT = 1 + KEY\RECEXT
```

```
3070    EXTENT(NEWEXT) = 1
3080    EXTTRANS = EXTTRANS-(OLDEXT<>NEWEXT)
3090    OLDEXT = NEWEXT
3100    GET #1,KEY
3101    LSET DAT$=MKI$(1+CVI(DAT$))
3102    PUT #1,KEY
3110    PRINT COUNT, KEY, NEWSEC, NEWEXT, CVI(DAT$)
3120 WEND
4000 PRINT : PRINT "Closing file"
4010 CLOSE #1
4020 PRINT CHR$(7);"File closed" :' stop the watch
5000 PRINT COUNT;" records processed of";NUMREC
5005 COUNT = 0
5010 FOR J = 1 TO NUMSEC
5020    COUNT = COUNT + SECTOR(J)
5030 NEXT J
5040 PRINT COUNT;" unique sectors visited of";NUMSEC
5050 COUNT = 0
5060 FOR J = 1 TO NUMEXT
5070    COUNT = COUNT + EXTENT(J)
5080 NEXT J
5090 PRINT COUNT;" unique extents visited of";NUMEXT
5100 PRINT SECTRANS;" sector transitions"
5110 PRINT EXTTRANS;" extent transitions"
9000 END
```

End Listing Four

Listing Five

```
1000 ' TREEUPDT.BAS: approximate a B+TREE update pattern
1005 RANDOMIZE(4093)    :' ensure repeatability
1010 DEFINT A-Z
1020 NUMBYTES = 128     :' bytes per record
1030 NUMREC = 1024      :' records in file (128*1024=128KB)
1040 RECSEC = 1024\NUMBYTES :' 1K sectors in use
1050 NUMSEC = NUMREC\RECSEC :' disk sectors in file
1060 RECEXT = 16384\NUMBYTES:' 16KB extents in use
1070 NUMEXT = NUMREC\RECEXT :' directory extents in file
2000 PRINT "Opening file";CHR$(7) :' start the watch
2010 OPEN "R",#1,"TEST.DIR",NUMBYTES
2020 FIELD #1, 2 AS KEY$, 2 AS DAT$, NUMBYTES-4 AS FILL$
3000 FOR COUNT = 1 TO 100
3010    K1 = INT(RND*8)+1
3020    GET #1,K1       :' top-level index read
3030    K2 = INT(RND*64)+9
3040    GET #1,K2       :' 2nd-level index read
3050    K3 = INT(RND*(NUMREC-73))+73
3060    GET #1,K3       :' data record
3070    LSET DAT$ = MKI$(CVI(DAT$))
3080    PUT #1,K3
3090    PRINT COUNT, K1, K2, K3
3100 NEXT COUNT
4000 PRINT : PRINT "Closing file"
4010 CLOSE #1
4020 PRINT CHR$(7);"File closed" :' stop the watch
9000 END
```

End Listing Five

Serial to Parallel
A Flexible Utility Box

by Alan D. Wilcox

What do you do when you have the chance to buy a working IBM Selectric I/O typewriter for a bargain price? Buy it, right? Especially when everything is all ready to plug in and run. During the happy trip home, you recall that the demo computer used a parallel port to drive the Selectric.

A parallel port on the demo computer? Minor oversight, you have a couple of spare serial ports and there shouldn't be a bit of trouble hooking up something to work right away. Guess again!

Sound like a familiar story? We've all been there somewhere along the way.

The Problem

I needed a serial-to-parallel converter – just a simple device to take the serial data from the RS-232C serial port on my Heath H-8 and convert it to parallel with strobe and handshaking. Because I wanted to have the flexibility of also using my H-89 with the Selectric, that excluded the option of actually building a parallel port into the computer itself. Whatever I decided to use, however, should not require special initialization or an extra driver program to make it run properly; it should be treated no differently than my serial-input Okidata 82A.

A Chain of Boxes

Figure 1 (page 31) gives an overview of the computer system. The computer's UART sends serial data to the serial-to-parallel box yet to be built. The parallel data from the box goes directly into the Selectric interface unit. The data, even at this point, are still plain ASCII; the code conversion to either Selectric Correspondence or BCD code takes place in the interface unit and needs no special attention. In the return direction, note that a busy signal comes from the interface unit when a character is received. The signal is passed on through to the computer's UART. Thus, the printing operation involves sending a single character, waiting for the Selectric to print it, sending another character, waiting, and so on.

Dr. Alan D. Wilcox, P.E., 60 South 8th St., Lewisburg, Pennsylvania 17837.

In Search of the Wheel

Rather than reinvent the wheel, I first checked the design Heath used[1] for their parallel interface board: two back-to-back 8251 USART's (Universal Synchronous/Asynchronous Receiver/Transmitters)! Not only was that more hardware than reasonable, but it also required a special driver program for initialization before it would even run at all. There had to be a better way.

A search of the literature turned up some useful ideas. Steve Ciarcia[2] published the schematic of a serial-to-parallel converter using the AY3-1015 UART (Universal Asynchronous Receiver/Transmitter) and NE-555 timer; while simple and quick to build, it did not have the flexibility of being able to easily change the baud rate. Two more of his designs[3,4] solved this data-speed flexibility problem by using a switch-settable baud-rate clock; these serial-parallel circuits were only a small part of much larger designs. In general, though, I liked his approach to data conversion.

Isenson[5] had a completely different solution to his parallel interface problem. Rather than deal with the serial data at all, he simply connected into his computer bus for the parallel data and used the computer's internal UART only for device decoding. As far as the computer was concerned, it was writing data to a serial port. He used the CPU's WAIT* (wait not) line to hold the CPU in a wait state until his printer was ready to accept more data. This approach would work in my application, but would hardly allow for the flexibility of hooking the Selectric to either the H-8 or the H-89.

Isenson did circumvent one critical design issue with my Selectric: handshaking. Pulling the WAIT* low when the printer is busy causes an immediate halt in CPU activity – not a bit more data gets sent to the printer until the WAIT* is released. On the other hand, in an application not using WAIT*, the computer's UART will finish sending the current character even though it receives a busy signal from the printer. Ordinarily this really is not too critical – most parallel printers on the market have buffering, so that some data can still be received even after a busy signal has been sent to the CPU's UART. Not so my Selectric. There is no buffer at all in the solenoid-driver/code-conversion circuit. This means that the sending UART in the computer has to receive a busy signal immediately after sending a character or else it will start sending a second character before the Selectric can accept it. Unfortunately, the second character has already started by the time the Selectric sends back its busy signal. Somehow, a pseudo-busy has to be sent to the CPU before its UART starts a second character.

What was thought to be a simple distraction is now getting to the level of a major project!

Circuit Design

The design of the completed serial-parallel converter is shown in Figure 2 (pages 32-33). The heart of the unit is the baud-rate clock and the UART; the rest is circuitry I found necessary to make the unit work with the Selectric. While designing it, I made the unit as general as possible so that it could be used with any computer with a serial port. In addition, I had in mind that my serial-parallel converter might also be used to get parallel data not only to a printer, but also to a PROM programmer sometime in the future.

The flow of serial data from the computer is from the DB-25 connector (pin 3), through the MC-1489 line receiver, and into the UART. When the UART has received a complete character, the DAV (Data Available) line at pin 19 goes high to signal that the parallel output is valid; the parallel output will remain valid until another character is received. In this design, the DAV signal is used for two purposes: the first is to trigger the 74123 one-shot so a strobe pulse goes out to the printer, and the second is to reset the UART DAV line by sending a low signal to the RDAV* (Reset Data Available not) input on pin 18. The normal sequence of operation, then, is that data come in serially until a complete character is received. This makes DAV true and triggers the strobe to the printer. At the same time as the strobe, RDAV* is activated to prepare the UART for a new character.

A word on the strobe might be in order at this point. The momentary high signal on DAV is very brief, somewhat less than a microsecond. My Selectric, and most other printers, need a microsecond or longer pulse duration for proper operation. The 74123 one-shot, as designed, provides a pulse on the order of 2 μsec. This has not proved to be critical, but it can be adjusted if desired by changing the resistor and capacitor (R6,C2) on pins 14

and 15 of IC7. The strobe itself is low-true.

One unexpected hitch arose in the development of the circuit: the Selectric has automatic line feed (LF). This meant that all I needed to send to the Selectric was a carriage return (CR) and it would go to the next line by itself. I could make allowance for this by changing the operating system to send just the CR character and not send the LF. However, the Okidata was set to receive the CR/LF pair and that's the way the Selectric better take it. (At this point, it was "Take it . . . or else.") My solution is the logic using the 7404 and 7430 IC's. When a line feed character (0AH) appears on the data bus from the UART, the logic combines to bring pin 8 of the 7430 low. If the switch is positioned to "delete LF," then pin 1 of the 7400 is pulled low, preventing the initiation of the strobe pulse. Without a strobe pulse, the data bus values are ignored by the printer and the undesired line feed gets no further.

The selection of the Western Digital BR1941 baud-rate clock was made primarily because of its availability and many switch-settable baud rates. There are actually several frequency options: the BR1941-00 and the BR1941-06 which both take the 5.0688 MHz crystal, and the BR1941-05 which takes a 4.9152 MHz crystal. The one sold by mail is the common BR1941-00. The IC can provide two different frequencies, one for a receiver and another for a transmitter. In the present design, however, the receiver clock is the only one used. Its output on pin 3 is at 16 times the baud rate selected by the switches. For example, if 1200-baud serial data are to be received, then the frequency of the signal at pin 3 is 16 x 1200 or 19.2 KHz. This clock signal goes directly into the UART RCP (Receiver Clock) on pin 17.

A decision needs to be made as to what data format is expected to be converted from serial to parallel. In my case, I expected to be receiving the same serial data as the Okidata, and that happened to be in the format of 1 stop bit, 8 bits per character, and no parity. All this needs to be set only one time and never needs changing unless the system is being altered. No special initialization data are required to be sent from the computer to make it run. This one-time setup is accomplished by the five switches connected to the UART pins 35-39 and will be discussed later.

The handshaking between the computer and the Selectric turned out to be easier than I first imagined. True, I did need to provide the pseudo-busy signal, as I call it, back to the CPU's UART before the Selectric even sent its busy signal, but it was simply done. Here's how it works: the instant the start bit for the first character comes in (on pin 3 of the DB-25 connector), the MC-1489 line receiver sends it to one section of the 74123 one-shot. The output of the one-shot (pin 12), which had been high until now, suddenly drops low and causes the output of the MC-1488 line driver to go high. In my system, a high signal (+12 volts) going out on the RTS (Request To Send, pin 4) is taken to mean that the printer is busy and cannot accept more data. Therefore, the high output of the MC-1488 causes the CPU's UART to see a busy on the handshaking line. This pseudo-busy appears during the first character quickly enough to prevent the CPU's UART from sending the next character; the busy stays on for about 28 msec. Meanwhile, as soon as the first character has ended, the busy line from the Selectric goes high. After being inverted in the 7400 to a low level, it prevents the MC-1488 line driver from changing its output (which is still high) to a low (-12 volts) not-busy. Once the Selectric busy has arrived, the one-shot can time-out and is no longer needed. At a typing rate of 12 characters per second, the total time is about 83 msec. for each character, so the arbitrary time for the one-shot can be quite flexible.

Construction

The whole prototype serial-to-parallel converter fits easily on one 4½-by-6-inch prototype board. A list of parts for the board is shown in Table I (page 34); The actual parts layout was not critical, and I positioned parts for simple interconnections. A wire-wrap approach could have been taken, but I preferred to just solder wires point-to-point – soldering the wires is a bit neater and easier to change. The board itself was mounted inside a walnut desk-top enclosure.

The cable going to the Selectric had a standard card-edge connector intended for a TRS-80 parallel port. I chose to provide a mate for it rather than change the connector itself. For the prototype, this involved etching a small piece of double-sided circuit board to match the pins. Recently, I noticed that the local Radio Shack store now has a prototype board with a suitable card-edge connector already etched on it. The serial data DB-25 connector was mounted to the circuit board with small angle brackets. The only other connections to the board are for power, and I used the card-edge connections on the other end of the board for them.

The BR-1941 requires two supply voltages for operation. Be sure not to apply one without the other. I accidentally turned off one supply, and the result was disaster for the baud-rate clock. The UART may require two supplies also, depending on which UART you use. The AY3-1014 I used needs only +5; others may need -12 on pin 2.

Rather than build a power supply, I chose one from the many on the market. The Micromint supply with +5 and ±12 was ideal for the job. The transformer is at the AC plug (like many calculator chargers), and the voltage regulation is on a small circuit board which I put in the same box as the converter itself.

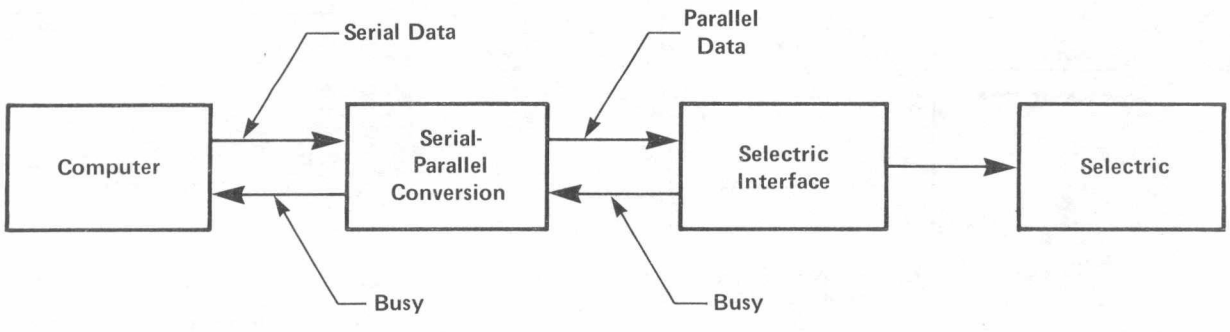


Figure 1.

Data Flow from the Computer to the Selectric.

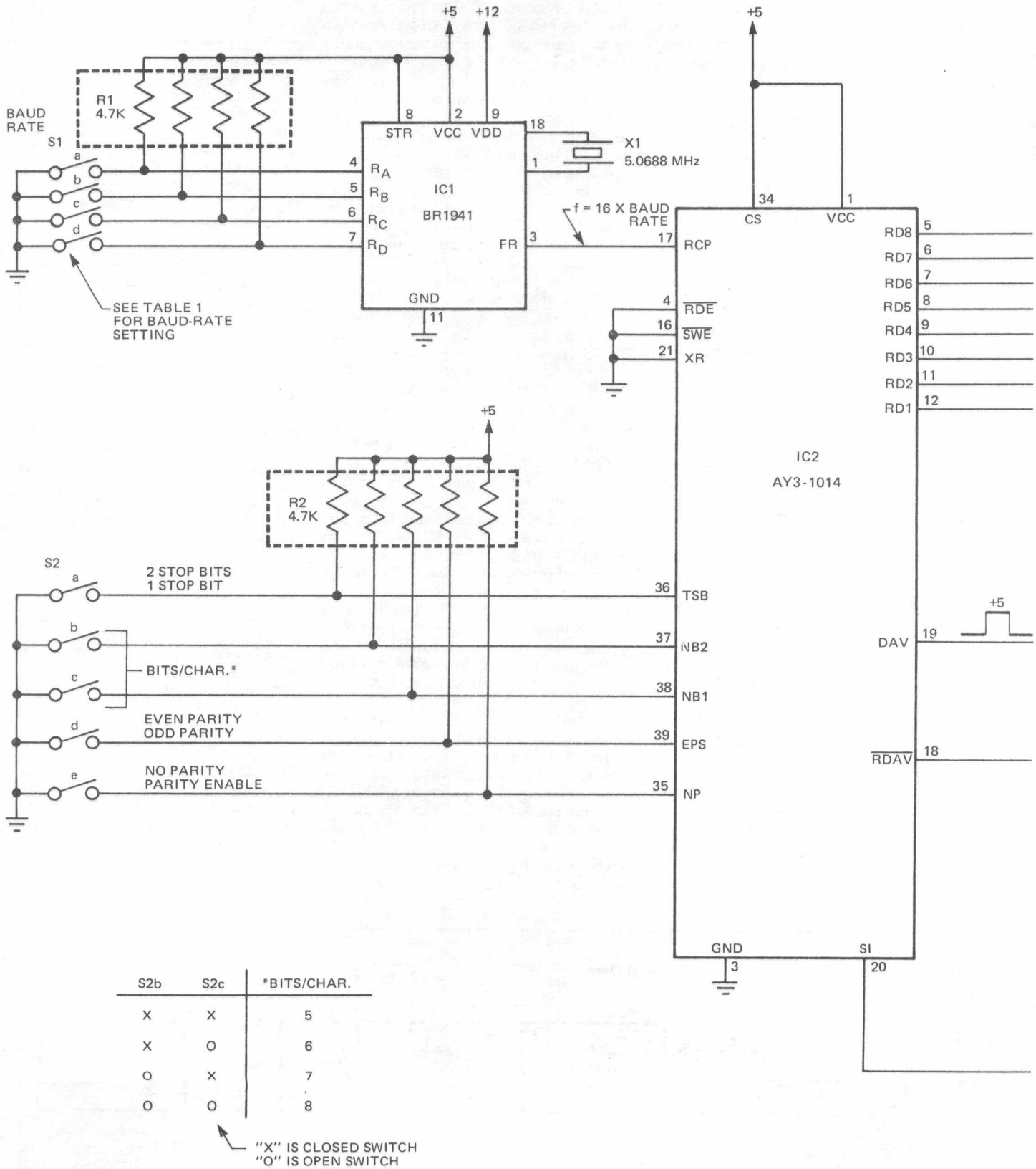
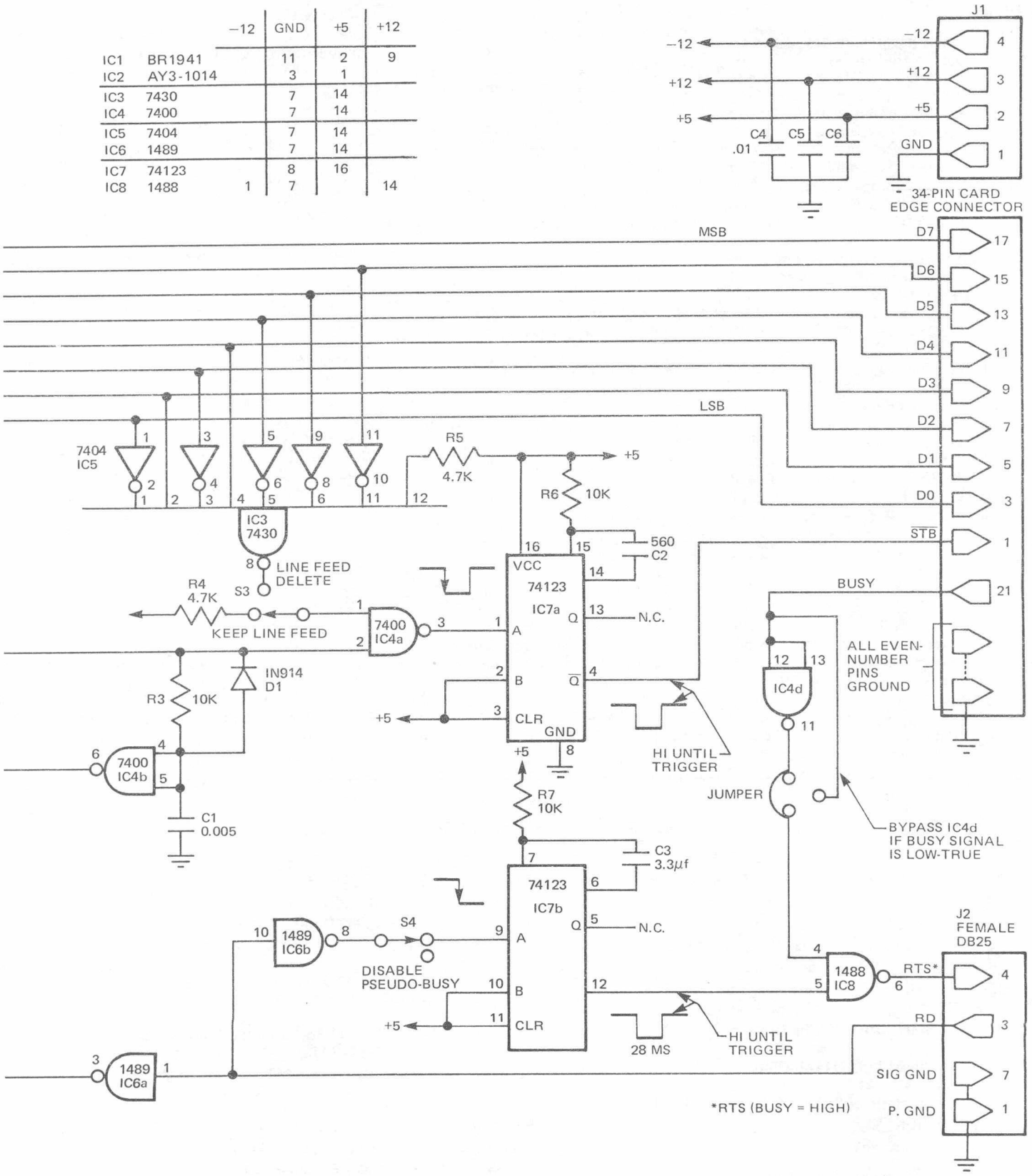


Figure 2.
SERIAL-PARALLEL CONVERTER

Alan D. Wilcox
5/7/83

		−12	GND	+5	+12
IC1	BR1941		11	2	9
IC2	AY3-1014		3	1	
IC3	7430		7	14	
IC4	7400		7	14	
IC5	7404		7	14	
IC6	1489		7	14	
IC7	74123		8	16	
IC8	1488	1	7		14



Initial Setup and Test

When first setting up the converter unit, make sure that the system is really what you think it is. Is RTS-high taken to mean a busy? Is the serial data at 1200 baud? Check by running CONFIGUR (or whatever program your particular operating system uses to set up serial ports). Be sure your cables are going to the right pins of the connectors. I can plug into two sockets on the back of the H-89: either the 340Q (my Okidata port) or the 320Q.

Be sure the converter is wired corectly. A safe check to make is to apply power with no IC's in their sockets; see if +5V and ±12V are where they should be. Install the IC's when it looks like the smoke test will be a happy one.

Set the baud-rate switches to 1200 baud: S1-a, -b, and -c should all be open. Set the other switches for the format I indicated earlier: S2-a closed for 1 stop bit, S2-b and S2-c open for 8 bits/character, S2-d can be open or closed because S2-e is set open for no parity. Table II (page 35) shows all the possible baud-rate selections; Table III (page 35) indicates how I set S1 and S2 for my system.

Connect the serial-data cable (leave the parallel cable off for now) and apply power. Check with your oscilloscope on pin 17 of the UART to see if the clock is present. Check pin 3 on the DB-25 connector: it should be -12V if you haven't sent anything down the line yet. Pin 4 should be -12V to indicate that the port is ready to take data.

If it appears everything is ready to go, send some data to the converter. The simplest data to check with the scope is a square-wave, and you can get that by sending a "U" to the port. Assuming you've connected into what your operating system takes as the printer port, do a simple BASIC program like this:

```
100  LPRINT "U";
200  GOTO 100
300  END
```

You should see the data coming in on pin 3 one character at a time, each separated from the next by about 30 msec. If you see a square-wave with no breaks between characters, then you know that the RTS line (or whatever busy line you're using) is not causing the CPU's UART to wait. Each of the characters should be 8.3 msec. total time and be an alternating sequence of +12V and -12V pulses. If you observed the proper handshaking, you can be sure that the computer and the converter are at least talking with each other. While the "U" is still being sent, look at the strobe output (a logic probe with memory is probably the easiest to check with). Did it catch a pulse? If not, see if you catch a pulse back at the UART pin 19. If not there either, be sure that RDAV* on pin 18 is not being held low for some reason.

Check the parallel output lines with the scope or logic probe. You should see alternating high and low levels when you look from one pin to the next. When you look at an individual pin, however, the logic level should not change (except perhaps once every 132 characters if the system puts a CR/LF in automatically). If you find that the data lines are rattling around in a somewhat random fashion, probably the baud rate is not matched to what the CPU is sending. Try other rate combinations or get an accurate frequency check of what the UART is getting on pin 17.

If you got this far, you probably have a working converter and you're ready to hook up the printer or whatever you want to put on the parallel port. Be sure that it has a +5V high-true line coming back for handshaking; if your device is low-true, skip around the 7400 inverter in the diagram. If you intend to have no handshaking at all, adjust the resistor and capacitor on pins 6 and 7 of the 74123 so that the data speed is acceptable for whatever device you have connected to the parallel port.

Operation

Once the converter has been tested and is running properly, it should need no further attention. I leave it plugged into the 320Q port until I want to use the Selectric. Then I tell the operating system that my printer is on 320 rather than 340. At that point, I treat it no differently than I would the Okidata. If I can't get back to the operating system conveniently, I can always just swap serial data cables.

Conclusions

Overall, I would say that the project has been an enjoyable experience for me. I've been very pleased with the converter's operation – it's everything I hoped it would be. There are only a few minor details that I might change if I were to build it over. One change might be to put

Table I
Parts List for Serial-To-Parallel Board

IC1	BR-1941-00	Western Digital Baud-Rate Clock
IC2	AY3-1014	General Instruments UART
IC3	7430	8-input NAND
IC4	7400	Quad 2-input NAND
IC5	7404	Hex Inverter
IC6	MC1489	Line Receiver
IC7	74123	Dual Monostable
IC8	MC1488	Line Driver
D1	IN914 or IN4148 Diode	
R1,R2	5-Resistor SIP 4.7K, Boumes 4306R-101-472	
R3,R6,R7	10K, ¼W	
R4,R5	4.7K, ¼W	
C1	0.005 Disk Cap 100 V	
C2	560 pF Sil. Mica	
C3	3.3 μF Tantalum 15V	
C4,C5,C6	0.01 Disk 100V	
S1	4-Pos. DIP SPST Switch	
S2	5-Pos. DIP SPST Switch	
S3	SPDT PCB MTD Slide Switch	
S4	SPST PCB MTD Slide Switch	
X1	5.0688 MHz Crystal (wire leads, series resonant, 0-70°C, ±0.01%, $R_s < 50\Omega$)	
J1	Male 4-Contact Power Plug (Molex Series 6373, No. 22-11-2042)	
J2	Female DB-25, PCB MTD, Right Angle (Cannon DBP-25-SAA)	
5 each	14-Pin Sockets (Solder-Tail)	
1 each	16-Pin Sockets (Solder-Tail)	
1 each	18-Pin Sockets (Solder-Tail)	
1 each	24-Pin Sockets (Solder-Tail)	
1 each	PC Board with 34-Pin Card-Edge Connector	

Table II
Baud-Rate Switch Settings

Switch S1 d c b a	Baud Rate
X X X X	50
X X X O	75
X X O X	110
X X O O	134.5
X O X X	150
X O X O	300
X O O X	600
X O O O	1200
O X X X	1800
O X X O	2000
O X O X	2400
O X O O	3600
O O X X	4800
O O X O	7200
O O O X	9600
O O O O	19200

Note: "X" means switch closed, "O" means open.

in a switch to disable the pseudo-busy circuit so it wouldn't hold up a full-speed printer. I drew S4 into the circuit where I would probably put it. A major modification to consider would be to increase the scope of the project by connecting up the other half of the BR1941 to the transmitter half of the UART, thereby being able to send different-speed serial data back to the CPU. If you have the need to convert parallel data to serial, this might be worth investigating.

As it turned out, getting the Selectric in operation was really quite easy . . . once I had my converter box built. Being able to simply plug in the utility box made a world of difference in making a smooth-running system.

For those of you who do not wish to put all this together for yourself, Mr. Wilcox's converter is available in various stages of completeness (from the PC board only, to an assembled and tested unit, minus case and power supply) from Micro Resources Inc., 9064 Loreleigh Way, Fairfax, Virginia 22031. Contact them for details.

References

[1] *Parallel I/O Interface, Model H8-2,* Manual 595-2033-04, Heath Company, Benton Harbor, MI, 1977.

[2] Ciarcia, Steve, ed., "Ask Byte," *BYTE,* p. 212, July 1981.

[3] Ciarcia, Steve, "Build an Unlimited-Vocabulary Speech Synthesizer," *BYTE,* pp. 38-50, September 1981.

[4] Ciarcia, Steve, *Ciarcia's Circuit Cellar, Vol. II,* pp. 203-217, Byte/McGraw Hill, Peterborough, NH, 1981.

[5] Isenson, Raymond S., "A Printer for the H-89," *Microcomputing,* pp. 144-148, October 1981.

Additional Reading

Flystra, Dan, "Interfacing the IBM Selectric Keyboard Printer," *BYTE,* pp. 46-52, 133-139, June 1977. (This is an excellent article covering Selectric operation, code-conversion, and interfacing.)

Table III
Example Setting for S1 and S2

1200 baud
1 stop bit
8 bits/character
No parity

S1

	on	off	
a		●	open
b		●	open
c		●	open
d	●		closed

S2

	on	off		
a	●		closed	1 stop bit
b		●	open	8 bits/char.
c		●	open	8 bits/char.
d		●	open	not in use*
e		●	open	no parity

*Can be open or closed with no parity select.

McWORDER
A Tiny Text Editor

by William A. McWorter

William A. McWorter, Jr., Mathematics Department, Ohio State University, 231 West 18th Avenue, Columbus, Ohio 43210.

I am writing this article with a 54-line (without comments) word processor. I find that it is nearly as much fun to use as it was to write. I call this editor, for obvious reasons, McWORDER. The source listing (pages 38-42) is written for the TRS-80 Model II, though it should not be too difficult to convert it for other machines. The program is not yet finished. The block-handling subroutines in the listing are not installed.

McWORDER functions in two modes, insert and change. The insert mode is indicated by a steady cursor and the change mode by a flashing cursor. In the insert mode, printable characters are inserted in the text at the cursor position. The text separates to accommodate them and words overfowing the line length are gathered in a new line.

The cursor movements available are: left, right, up, and down. In the change mode all moves stay within text bounds. In the insert mode, "cursor up" creates a new line when the cursor is on the first line, and the remaining cursor moves may position the cursor one character position beyond the last character of a line. The text will scroll up or down as required by the cursor moves.

A control code deletes a character at the current cursor position and the current line closes up behind it. If the cursor is at the beginning of a line and the line is empty, the line will be deleted and the text will close up. All text changes are shown on the screen because the print subroutine always prints what is in the text. Even scrolling uses the print subroutine. There is no separate scrolling subroutine. Action is the same in both modes.

Backspace replaces the previous character by a space and moves the cursor one character left when in the change mode. When the cursor is at the beginning of a line, backspace erases the last character of the previous line. In the insert mode, backspace deletes the previous character and the current line closes up. If the cursor is at the beginning of a line, and if the line is empty, backspace deletes the line, the text closes up, and the cursor is positioned just after the last character of the previous line.

Carriage return puts the cursor at the beginning of the next line. In the insert mode, the next line is a new line.

There are control codes to output the text to the printer, save the current text to disk, and load a text file from disk.

Finally, there are control codes specific to the Model II that are used by the subroutines. 1 turns on the blinking cursor, 2 turns the cursor off, 4 turns on the steady cursor, 23 clears to the end of the line, and 24 clears to the end of the screen.

The data structure used for the text is a doubly linked list whose pointers are in separate arrays L(.) and R(.). I have arranged that the list be circular with the left link of the first line pointing to the last line and the right link of the last line pointing to the first line. L(0) and R(0) contain the locations of the first and last lines, respectively. All text is constrained to have at least one line. New text begins with one empty line and all pointers point to this empty line.

My purpose in writing this editor had been to familiarize myself with handling linked lists. However, it turned out that the most challenging problem was designing efficient control structures (well-chosen and positioned IF . . . THENs). I am still studying how best to handle them.

Writing McWORDER has been a frustrating, excruciating, exasperating delight. I hope that *DDJ* readers will find parts of the editor useful for other editing tasks and, just maybe, will be inspired, as I have been, to modify and improve upon the editor – a labor of love possible only when commented source code invites the user.

DDJ

(Listing begins on page 38)

McWorder Listing (Text begins on page 36)

```
10 'MCWORDER: A TEXT EDITOR
20 '
30 'TEXT IS HELD IN STRING ARRAY T$(.)
40 'LINKS ARE HELD IN L(.) AND R(.)
50 'L(0)=INDEX OF FIRST LINE, R(0)=INDEX OF LAST LINE.
60 'N=INDEX OF CURRENT LINE.
70 '
80 'INITIALIZE EDITOR (RO,CO)=TEXT SCREEN DIMENSIONS
90 '
100 CLEAR 20672:DEFINT A-Z:RO=24:CO=80:DIM T$(100),A(20),
     L(100),R(100):S=0:R=5:C=0
110 '
120 '28=CURSOR LEFT; 29=CURSOR RIGHT; 30=CURSOR UP;
     31=CURSOR DOWN
130 '1=TOGGLE INSERT MODE; 2=DELETE CHAR; 8=BACKSPACE;
     13=CARRIAGE RETURN
140 '16=OUTPUT TEXT TO PRINTER; 4=SAVE TEXT TO DISK;
     12=NEW TEXT OR LOAD TEXT
150 '
160 DATA 28,29,30,31,1,2,8,13,16,4,12
170 FOR I=1 TO 11:READ X:D$=D$+CHR$(X):NEXT I
180 '
190 'GET TEXT FROM DISK OR WRITE NEW TEXT
200 '
210 GOSUB 1380
220 '
230 'GET CHARACTER FROM KEYBOARD AND PROCESS IT
240 '
250 PRINT@(R,C),;:A$=INPUT$(1):IF A$>CHR$(31) THEN
     GOSUB 1050:GOTO 250
260 ON INSTR(D$,A$) GOSUB 750,800,850,910,960,1000,
     1160,1220,1270,1330,1380
270 GOTO 250
280 '
290 '--------------- SUBROUTINES ---------------
300 '
310 'INSERT NEW LINE B$ AFTER LINE INDEXED N
320 '
330 IF S THEN T=A(S):S=S-1 ELSE T=M:M=M+1
340 T$(T)=B$:IF N=R(0) THEN R(0)=T
350 L(T)=N:R(T)=R(N):L(R(T))=T:R(N)=T:RETURN
360 '
370 'INSERT NEW LINE B$ BEFORE LINE INDEXED N
380 '
390 IF S THEN T=A(S):S=S-1 ELSE T=M:M=M+1
400 T$(T)=B$:IF N=L(0) THENL(0)=T
410 R(T)=N:L(T)=L(N):R(L(T))=T:L(N)=T:RETURN
```

(Continued on next page)

McWorder Listing (Listing continued, text begins on page 36)

```
420 '
430 'DELETE LINE INDEXED F
440 '
450 IF F=R(0) THEN IF F=L(0) THEN RETURN ELSE R(0)=L(F) ELSE
     IF F=L(0) THEN L(0)=R(F)
460 S=S+1:A(S)=F:R(L(F))=R(F):L(R(F))=L(F):RETURN
470 '
480 'DELETE LINES BETWEEN LINE INDEXED F AND LINE INDEXED
     F1
490 '
500 T=F:IF F1=R(0) THEN R(0)=L(F)
510 S=S+1:A(S)=T:IF T=F1 THEN R(L(F))=R(F1):L(R(F1))=L(F):
     RETURN
520 T=R(T):GOTO 510
530 '
540 'MOVE BLOCK START INDEX F, END INDEX F1, TO AFTER LINE
     INDEXED N
550 '
560 IF N=R(0) THEN R(0)=F1 ELSE IF F1=R(0) THEN R(0)=L(F)
570 IF F=L(0) THEN L(0)=R(F1)
580 R(L(F))=R(F1):L(R(F1))=L(F):L(R(N))=F1:R(F1)=R(N):
     R(N)=F:RETURN
590 '
600 'MOVE BLOCK START INDEX F, END INDEX F1, TO BEFORE LINE
     INDEXED N
610 '
620 IF N=L(0) THEN L(0)=F ELSE IF F=L(0) THEN L(0)=R(F1)
630 IF F1=R(0) THEN R(0)=L(F)
640 R(L(F))=R(F1):L(R(F1))=L(F):R(L(N))=F:L(F)=L(N):
     L(N)=F1:R(F1)=N:RETURN
650 '
660 'PRINT TEXT FROM SCREEN LINE B TO END OF TEXT OR SCREEN
670 'STARTING FROM LINE INDEXED T
680 '
690 PRINT@(B,0),CHR$(2);:FOR X=B TO RO-1:PRINT@(X,0),CHR$(23);
     T$(T);
700 IF T=R(0) THEN PRINTCHR$(24);:PRINT@(R,C),
     CHR$(4+3*(I=0));:RETURN
710 T=R(T):NEXT X:PRINT@(R,C),CHR$(4+3*(1=0));:RETURN
720 '
730 'CURSOR LEFT
740 '
750 IF C>0 THEN C=C-1:RETURN ELSE IF R>5 THEN R=R-1 ELSE
     IF N<>L(0) THEN FS=L(N):B=5:T=FS:GOSUB 690 ELSE RETURN
760 N=L(N):C=LEN(T$(N))+(LEN(T$(N))>0 AND 1=0):RETURN
770 '
780 'CURSOR RIGHT
790 '
```

```
800 IF C<LEN(T$(N))+(I=0) THEN C=C+1:RETURN ELSE
    IF N=R(0) THEN RETURN
810 N=R(N):IF R=RO-1 THEN B=5:FS=R(FS):T=FS:C=0:
    GOTO 690 ELSE R=R+1:C=0:RETURN
820 '
830 'CURSOR UP
840 '
850 IF R>5 THEN R=R-1:N=L(N):
    T=LEN(T$(N))+(I=0 AND LEN(T$(N))>0):C=C+(C-T)*(C>T):
    RETURN
860 IF N=L(0) THEN IF I THEN B$="":GOSUB 390 ELSE RETURN
870 B=5:FS=L(N):T=FS:GOSUB 690:N=L(N):
    T=LEN(T$(N))+(I=0 AND LEN(T$(N))>0):C=C+(C-T)*(C>T):
    RETURN
880 '
890 'CURSOR DOWN
900 '
910 IF N=R(0) THEN RETURN ELSE IF R=RO-1 THEN B=5:FS=R(FS):
    T=FS:GOSUB 690 ELSE R=R+1
920 N=R(N):T=LEN(T$(N))+(I=0 AND LEN(T$(N))>0):
    C=C+(C-T)*(C>T):RETURN
930 '
940 'TOGGLE INSERT MODE
950 '
960 I=-(I<1):PRINT@(R,C),CHR$(4+3*(I=0));:RETURN
970 '
980 'DELETE CHARACTER AT CURSOR
990 '
1000 IF T$(N)="" THEN F=N:IF N<>R(0)
    THEN N=R(N):GOSUB 450:B=R:T=N:GOTO 690 ELSE GOSUB 750:
    GOTO 450
1010 C=C+(LEN(T$(N))=C):
    T$(N)=LEFT$(T$(N),C)+MID$(T$(N),C+2):
    PRINT@(R,C),CHR$(23);MID$(T$(N),C+1);:RETURN
1020 '
1030 'CHANGE OR INSERT CHARACTER
1040 '
1050 IF I=0 THEN IF T$(N)>"" THEN MID$(T$(N),C+1)=A$:
    PRINT@(R,C),CHR$(2);A$;CHR$(1);:GOTO 800 ELSE RETURN
1060 IF T$(N)="" THEN T$(N)=A$:PRINT@(R,C),A$;:GOTO 800
1070 IF LEN(T$(N))<LL THEN
    T$(N)=LEFT$(T$(N),C)+A$+MID$(T$(N),C+1):
    PRINT@(R,C),CHR$(2);MID$(T$(N),C+1);CHR$(4);:GOTO 800
1080 L=LL+1:C$=LEFT$(T$(N),C)+A$+MID$(T$(N),C+1):
    FOR J=1 TO LEN(C$)
1090 IF INSTR(RIGHT$(C$,J)," ")=0 THEN NEXT J:J=1
1100 T$(N)=LEFT$(C$,L-J):B$=RIGHT$(C$,J-1):IF C>L-J-1 THEN
    FF=1:C=C-L+J
```

(Continued on page 42)

McWorder Listing (Listing continued, text begins on page 36)

```
1110 IF FI<>N THEN GOSUB 330:FI=N ELSE IF N<>R(0) THEN
     T$(R(N))=LEFT$(B$+" "+T$(R(N)),LL) ELSE GOSUB 330:
     T$(R(N))=B$
1120 B=R:T=N:GOSUB 690:IF FF THEN FF=0:GOTO 910 ELSE 800
1130 '
1140 'BACKSPACE
1150 '
1160 IF I=0 THEN 1180 ELSE IF C THEN GOSUB 750:
     GOTO 1000 ELSE IF T$(N)>"" THEN GOSUB 750:GOTO 1000
1170 F=N:IF N<>R(0) THEN N=R(N):GOSUB 450:B=R:T=N:GOSUB 690:
     GOTO 750 ELSE GOSUB 750:GOTO 450
1180 GOSUB 750:IF T$(N)>"" THEN MID$(T$(N),C+1,1)=" ":
     PRINT@(R,C)," ";:RETURN ELSE IF N=L(0) THEN RETURN ELSE
     1180
1190 '
1200 'CARRIAGE RETURN
1210 '
1220 IF I=0 THEN C=0:GOTO 910
1230 B$="":GOSUB 330:B=R:T=N:GOSUB 690:GOTO 910
1240 '
1250 'OUTPUT TEXT TO PRINTER
1260 '
1270 T=L(0)
1280 LPRINTT$(T):IF T=R(0) THEN RETURN
1290 T=R(T):GOTO 1280
1300 '
1310 'SAVE TEXT TO DISK
1320 '
1330 PRINT@0,CHR$(23);"TEXT NAME";:INPUT F$:OPEN"O",1,F$:
     T=L(0):PRINT#1,LL
1340 PRINT#1,T$(T):IF T=R(0) THEN CLOSE:RETURN ELSE T=R(T):
     GOTO 1340
1350 '
1360 'LOAD TEXT FROM DISK OR START NEW TEXT
1370 '
1380 FI=0:S=0:R=5:C=0:PRINT@0,CHR$(23);:
     INPUT"TEXT NAME (<CR> IF NEW TEXT)";Q$:IF Q$="" THEN
     1420
1390 OPEN"I",1,Q$:Q$="":INPUT#1,LL:FOR T=1 TO 101
1400 IF EOF(1) THEN R(0)=T-1:L(0)=1:L(1)=R(0):R(R(0))=1:N=1:
     M=R(0)+1:FS=N:I=0:B=5:T=1:CLOSE:GOTO 690
1410 LINEINPUT#1,T$(T):L(T)=T-1:R(T)=T+1:NEXT T:CLOSE:RETURN
1420 CLS:PRINT@0,;:INPUT"LINE LENGTH";LL:IF LL>CO THEN
     LL=CO:PRINT@0,"CHOPPED TO";CO;"CHARACTER LINE"
1430 T$(1)="":N=1:M=2:FS=1:I=1:L(0)=1:R(0)=1:L(1)=1:R(1)=1:
     PRINTCHR$(4);:RETURN
```

End Listing

Fast Matrix Operations
In Forth, Part III

This final part of the series uses the material presented in the first two parts to build some applications. In those earlier installments, the author used a word, IN#, which allows free-format numeric input. The Forth code for this word is provided in Screen No. 129 (page 47). While it is not part of Polyforth, it was written for the author by Forth Inc. and is printed here with the permission of Forth Inc. – Ed.

In the last two parts (*DDJ* Nos. 80 and 81), I presented some Forth words to perform matrix operations with syntactically simple phrases. In this, the last installment, these words will be used in some application programs to demonstrate the merits of such an extension to the Forth language. Also, these programs were selected to be interesting. It is hoped that incredulous readers will be tempted to duplicate some of these programs in their favorite language and thereby convince themselves of Forth's potential superiority as a number cruncher.

Image Processing

Matrix operations are used extensively in the field of Digital Image Processing. This usage is not at all surprising when one considers that a digital image is, itself, a matrix. Screens 294 and 296 (pages 48-50) contain a program, OUTPIC, that prints a "poor man's grey level picture" on the IBM (Epson) printer. If we store a picture in matrix A so that the value of each matrix element represents the relative brightness of a pixel, the phrase, PRINT OUTPIC, will print out the picture by using ASCII characters with overstrikes to achieve 32 brightness or grey levels. Figures 1a, 1b, and 1c (page 45) were produced by OUTPIC. Figure 1a is titled "Smiley Face" and Figure 1b is titled "Snow Storm." These images were chosen because they are easily generated.

Consider the following problem: Given two pictures, Figures 1a and 1b, produce a third picture (Figure 1c), which has the image of 1a and the grey level histogram of 1b. In other words, how do we rearrange the pixels of 1b to make 1b look like 1a? In the simple Rosenfeld method, the darker pixels of 1b are given priority assignment to the locations of the darker pixels of 1a. Generally, in this method, a random choice is made in the assignment of pixels of equal grey levels, producing a blotchy result. In the program of screens 297 and 298 (pages 50-51), MATCH-A1-to-A2, priority assignment of pixels with equal grey level values is made on the basis of their closeness to other dark pixels. Figure 1c is the result of this histogram matching technique.

by Steven A. Ruzinsky

Steven A. Ruzinsky, 2110 South Austin Boulevard, Cicero, Illinois 60650.

Least Squares Regression

One of the most interesting and useful applications of matrices is Least Squares Regression analysis. Here, a functional relationship between a dependent variable, y, and n independent variables, $x_1, x_2, \ldots, x_n$, is modeled by the following equation:

$$(1)\quad y = a_1x_1 + a_2x_2 + \ldots + a_nx_n + E$$

in matrix notation: $y = \underline{a}^T\underline{x} + E$

where: $\underline{a} \equiv \begin{bmatrix} a_1 \\ a_2 \\ . \\ . \\ a_n \end{bmatrix}$, $\underline{x} \equiv \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix}$ and E = Error

The term, E, can be considered to represent everything that influences y that has not been accounted for by $\underline{x}$. In practice, the factors of E are usually unobservable; therefore, the equation actually used to approximate or estimate y from measured values of $\underline{x}$ is:

$$(2)\quad \hat{y} = a_1x_1 + a_2x_2 + \ldots + a_nx_n$$

or $\hat{y} = \underline{a}^T\underline{x}$

Hence, it is desirable to find $\underline{a}$ such that the error, E, is in some way minimized. In least squares regression, sample values of $\underline{x}$ and y are analyzed, and $\underline{a}$ is found such that the mean squared value of E is a minimum. There are several algorithms for performing least squares regression. Implementations of two such algorithms are presented here in Forth.

The first is a *"batch" least squares algorithm,* so called because all of the data is processed at one time. The matrix equation describing the algorithm is:

$$(3)\quad \underline{a} = (X^TX)^{-1}\ X^T\underline{y}$$

where: $X \equiv \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ . & . & \ldots & x_{3n} \\ . & . & \ldots & . \\ x_{r1} & x_{r2} & \ldots & x_{rn} \end{bmatrix}$

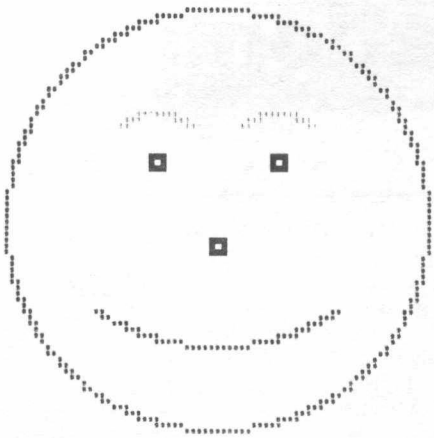
and $\underline{y} \equiv \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_r \end{bmatrix}$

Here, the subscript, i, of x_{ij} and y_i refers to different sets of measurements, r in number, and the subscript, j, of x_{ij} refers to the different variables, n in number. It is essential that $r \geqslant n$.

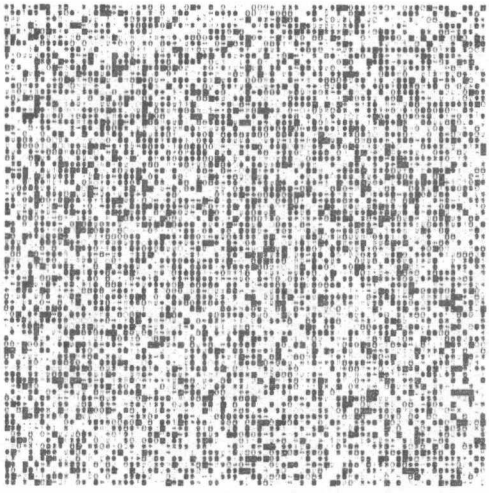
What do you do if you're on a deserted island with your computer and you need a 12th order polynomial approximation for a cosine but you don't have a handbook to look it up? Well obviously you let $x_1 = 1$, $x_2 = x$, $x_3 = x^2$, $x_4 = x^3$ etc. and use equation (3) to find $\underline{a}$, the polynomial coefficients. A program for doing this is in screen 301 (page 52). A printout of the results is given in Figure 2 (page 46). It is interesting to compare these results with those given in the *Handbook of Mathematical Functions* by Abramowitz and Stegun. The largest order polynomial given is 10th order and has a maximum absolute error of 2.0E-9. Here, my 12th order polynomial resulted in a maximum absolute error of 2.1E-12 – not bad for homebrew polynomial approximations.

In the preceding example, the error, E, represents the contribution to y of higher powers of x. If an infinite order polynomial were fitted to the data, E would be 0, and $\underline{a}$ would be the coefficients of the Tailor series expansion for the cosine. Frequently E represents random factors. The program given in screen 300 (page 52) is similar to the preceding except a random number has been added to the data. This program plots the data with its polynomial approximation. The results are shown in Figure 3 (page 46).

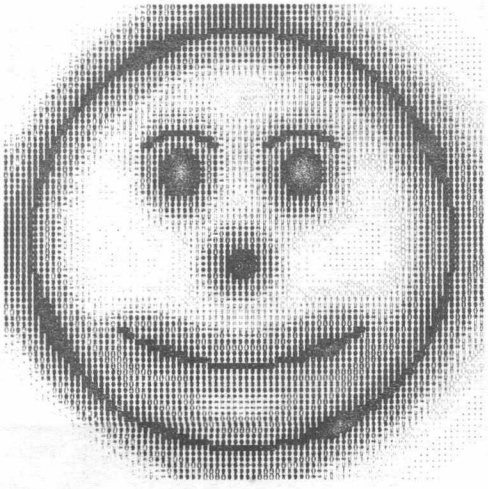
Next comes a *"sequential" least squares algorithm.* Let's say we have a constant input of data and we always want to make the best possible estimate of y with the data we have. As we receive new data, it would be best from the viewpoint of saving both speed and memory if we could base our calculations on old



(a)



(b)



(c)

Figure 1.

Demonstration of histogram matching of digital images. (a) is called "Smiley Face," (b) is "Snow Storm," and (c) shows Smiley Face matched to the histogram of Snow Storm.

```
a( 0,0) =   0.9999999999987197
a( 1,0) =  -0.0000000000000006
a( 2,0) =  -0.4999999999537521
a( 3,0) =   0.0000000000000042
a( 4,0) =   0.0416666663880298
a( 5,0) =  -0.0000000000000085
a( 6,0) =  -0.0013888882583870
a( 7,0) =   0.0000000000000070
a( 8,0) =   0.0000248009073781
a( 9,0) =  -0.0000000000000026
a(10,0) =  -0.0000002751960534
a(11,0) =   0.0000000000000003
a(12,0) =   0.0000000019835358

Maximum Absolute Error =
                0.0000000000020661
```

Figure 2.

Coefficients of 12th order polynomial approximation to cosine function found by least squares regression program of screen 301.

results and new data rather than old data and new data. The following algorithm provides a method for doing this:

$$(4a)\quad P_r = P_{r-1} - \frac{P_{r-1}\,\underline{x}_r\,\underline{x}_r^T\,P_{r-1}}{1 + \underline{x}_r^T\,P_{r-1}\,\underline{x}_r}$$

$$(4b)\quad \underline{a}_r = \underline{a}_{r-1} + P_r\,\underline{x}_r\,(y_r - \underline{x}_r^T\,\underline{a}_{r-1})$$

Here, the subscript, r, refers to the rth successive iteration. $\underline{x}_r$ is the rth measurement of $\underline{x}$. The initial value of P_r, i.e., P_o, should be set equal to zI where z is the largest number your computer can comfortably handle. $P_r = (X^TX)^{-1}$ of the batch algorithm and is therefore an nth order square matrix.

Suppose you want to predict tomorrow's stock price from past stock prices. After some preconditioning of the data you may be able to model the relationship as:

$$(5)\quad x_k = a_1x_{k-1} + a_2x_{k-2} + \ldots + a_nx_{k-n} + w_k$$

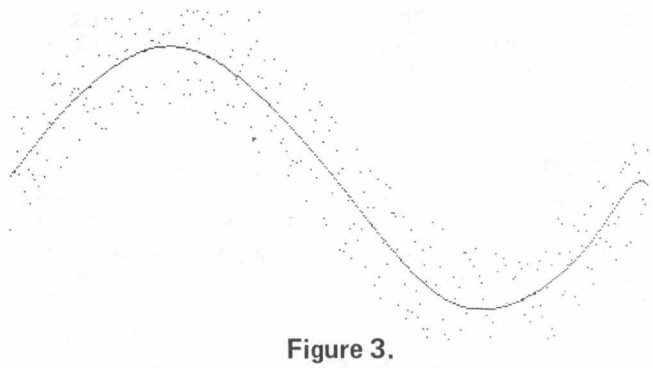


Figure 3.

Graph of scattered data with polynomial approximation produced by Program in screen 300.

Figure 4.

Autoregressive Signal (top) and its Estimate (bottom)

Here, x_k is tomorrow's price, x_{k-1} is today's, x_{k-2} is yesterday's, etc. w_k is a random number and therefore, by definition, unpredictable. If the coefficients, $\underline{a}$, were known, you could predict with error, w_k, tomorrow's stock prices by using:

$$(6) \qquad \hat{x}_k = a_1 x_{k-1} + a_2 x_{k-2} + \ldots + a_n x_{k-n}$$

In case you haven't noticed, equations (5) and (6) are just special cases of equations (1) and (2), so we can use least squares regression to find the coefficients, a.

In the programs of screens 302 and 303 (pages 52-53), a signal is generated directly from equation (5) with five autoregressive parameters, $a_1 = 0.6$, $a_2 = -0.4$, $a_3 = -0.2$, $a_4 = -0.3$, and $a_5 = -0.1$. A bar graph of 960 iterations of x_k is given at the top of Figure 4 (page 46) and its estimate, $\hat{x}_k$ is shown underneath. The program, DEMO, in screen 303 uses the sequential least squares algorithm to find $\underline{a}$ from $[x_k]$. A printout of the results of DEMO is given in Figure 5 (page 47).

In summary, with the advent of numeric co-processors, particularly those with stack architectures – e.g., the Intel 8087 and 80287 and upcoming Motorola 68881 (complement to the 68000) – Forth is potentially the best high-level language for utilizing their speed.

A major step in realizing this potential has been made by Forth, Inc. with their use of the 8087 register stack as an extra Forth stack in Polyforth Level 2 for the IBM PC. Hopefully, this article has indicated direction for further development of this potential. By extending Forth to include words similar to, but not identical to, those of APL (A Programming Language), Forth can provide the best combination of speed, interactivity, and programming convenience.

As a guideline for future development of matrix operations in Forth, it is recommended that the Forth words be more specialized and, hence, less versatile but more numerous, than those of APL. Tradeoffs of speed for error checking, versatility, and compactness should be minimized. The importance of an interactive approach in number crunching can not be overemphasized, and speed is necessary for interactivity.

So who says Forth is no good for number crunching? ▶▶J

(Listing begins below)

10. Iterations

a(0,0) = 0.1233901268682028
a(1,0) = -0.1245666093478517
a(2,0) = -1.2743755682353395
a(3,0) = -0.5459890505070130
a(4,0) = -0.2063259146671595

1000. Iterations

a(0,0) = 0.6266869267910911
a(1,0) = -0.4646612542696020
a(2,0) = -0.1753721863446832
a(3,0) = -0.3044790798136833
a(4,0) = -0.1227354803740424

100000. Iterations

a(0,0) = 0.6002403624134816
a(1,0) = -0.4029950413202153
a(2,0) = -0.1984026117414880
a(3,0) = -0.3019761820629188
a(4,0) = -0.0999649707415213

100. Iterations

a(0,0) = 0.3611617403734882
a(1,0) = -0.2500877037073712
a(2,0) = -0.1941419095476008
a(3,0) = -0.3755920541195973
a(4,0) = -0.1781407635452931

10000. Iterations

a(0,0) = 0.6067189034183671
a(1,0) = -0.4254754709960817
a(2,0) = -0.1729774822474928
a(3,0) = -0.3158399892817091
a(4,0) = -0.1012047713266529

1000000. Iterations

a(0,0) = 0.6014009069329175
a(1,0) = -0.4002088620556101
a(2,0) = -0.2003490862724384
a(3,0) = -0.2999269894431592
a(4,0) = -0.0991704866903059

Figure 5.

Estimates of autoregression coefficients of signal shown in Figure 4, found by program in screens 302 and 303.

Fast Matrix Operations (Text begins on page 44)

(Screen 129 printed with the permission of Forth Inc.)

```
129
 0 ( 8087 Free Format Numeric Input by Robert Drap of Forth, Inc.)
 1 CODE RESCAN ( a - a') 0 POP 0 0 OR 0= NOT IF  1 POP
 2    PTR U) INC 4 # I SUB   THEN NEXT
 3
 4 : >SCAN ( a - a')   -1025 PTR ! BEGIN ?DIGIT RESCAN DUP C@
 5    DUP 43 48 WITHIN SWAP 58 = + WHILE 0 PTR ! REPEAT ;
 6
 7 : SCAN ( a - t)    >SCAN DUP C@ DUP 32 - IF DUP 69 = SWAP 101
 8    = + ?DUP 0= IF DROP 0 EXIT THEN + >SCAN C@ 32 -
 9    IF 0 EXIT  THEN 1 PTR ! ELSE 2DROP THEN 1 ;
10
11 : IN# ( - t) ( - r)  QUERY 32 WORD DUP SCAN IF NUMBER
12    PTR @ 0< IF >N ELSE PTR @ 0= IF 2>N THEN THEN 0
13    ELSE 1 THEN 0 >IN ! 0 CNT ! DROP ;
14 : #IN IN# N> ;
```

(Continued on next page)

Fast Matrix Operations (Listing continued, text begins on page 44)

```
294 LIST

 0 ( Subroutines Used by OUTPIC ) 80 80 matrix A 80 80 matrix B
 1 : 0) 32 EMIT ; : 1) 46 EMIT ; : 2) 44 EMIT ; : 3) 45 EMIT ;
 2 : 4) 43 EMIT ; : 5) 111 EMIT ; : 6) 48 EMIT ; : 7) 79 EMIT ;
 3 : 8) 42 EMIT ; : 9) 36 EMIT ; : 10) 35 EMIT ; : BS 8 EMIT ;
 4 : 11) 10) BS 1) ; : 12) 10) BS 2) ; : 13) 10) BS 3) ;
 5 : 14) 10) BS 4) ; : 17) 10) BS 5) BS 7) ; : 15) 10) BS 6) ;
 6 : 16) 10) BS 7) ; : 18) 10) BS 8) ; : 19) 10) BS 9) ;
 7 : 20) 19) BS 1) ; : 21) 19) BS 2) ; : 22) 19) BS 3) ;
 8 : 24) 19) BS 4) ; : 25) 19) BS 5) ; : 26) 19) BS 7) ;
 9 : 27) 26) BS 26) ; : 23) 19) BS 8) ; : 28) 27) BS 27) ;
10 : 29) 175 EMIT BS 175 EMIT ; : 30) 29) BS 29) ; : 31) 30) BS
11 30) BS 30) ; : B) 16) ; : C) 17) ;
12 : 16) 19) ; : 17) 20) ;  : 19) B) ; : 20) C) ;
13 ' 31) ' 30) ' 29) ' 28) ' 27) ' 26) ' 25) ' 24) ' 23) ' 22)
14 ' 21) ' 20) ' 19) ' 18) ' 17) ' 16) ' 15) ' 14) ' 13) ' 12)
15 ' 11) ' 10) ' 9) ' 8) ' 7) ' 6) ' 5) ' 4) ' 3) ' 2) ' 1) ' 0)
```

```
295 LIST

 0 ( Line 1 transfers the addresses from the stack to array XX.   )
 1 32 ARRAY XX : FILL-XX 32 0 DO I XX ! LOOP ; FILL-XX
 2 : PRINT-GREY   XX @EXECUTE ; : COMP 27 EMIT 51 EMIT 22 EMIT ;
 3 variable MIN  ~ : MIN! MIN !1 ;  ~ : MIN@ MIN @1 ;
 4 variable MAX  ~ : MAX@ MAX @1 ;  ~ : MAX! MAX !1 ;
 5 : SELECT-MAX-MIN dim@ n! m! 0 0 A @1 FDUP MIN! MAX!
 6                  n@ 0 DO m@ 0 DO I J A @1 FDUP
 7                  MIN@ F< IF MIN! ELSE FDUP MAX@ F> IF MAX!
 8                  ELSE FDROP THEN THEN LOOP LOOP MAX@ MIN@ ;
 9 : QUANTIZE SELECT-MAX-MIN F- FDUP F0= NOT IF 31.0 FSWAP F/ MAX!
10            n@ 0 DO m@ 0 DO I J A DUP @1 MIN@ F- MAX@ F*
11            N> >N !2 LOOP LOOP ELSE FDROP THEN ['] A exc12 ;
12 ( OUTPIC is the poor man's grey level picture.              )
13 ~ : OUTPIC  ['] A QUANTIZE  m@ 0 DO n@ 0 DO J I
14             A @1  N>  PRINT-GREY  LOOP
15             13 EMIT COMP 10 EMIT LOOP 27 EMIT 64 EMIT ;
```

(Continued on page 50)

Fast Matrix Operations (Listing continued, text begins on page 44)

```
296 LIST

 0 ( Generate smiley-face for demonstration of MATCH                    )
 1
 2 VARIABLE XO VARIABLE YO VARIABLE R VARIABLE U1 VARIABLE U2
 3 : CIRCLE a! U2 ! U1 ! R ! YO ! XO ! U2 @ 1+ U1 @ DO I >N
 4 .0174533 F* FDUP SIN FSWAP COS R @ >N F* N> XO @ +
 5 R @ >N F* N> YO @ + SWAP a@ EXECUTE LOOP ;
 6 : Q .5 A !1 ; : W 1.0 A !1 ; : Z .3 A !1 ;
 7 : SMILEY-FACE ['] A clr
 8                39 39 35 0 360 ['] Q CIRCLE
 9                39 43 1 0 360 ['] W CIRCLE
10                49 29 1 0 360 ['] W CIRCLE
11                29 29 1 0 360 ['] W CIRCLE
12                49 29 8 225 315 ['] Z CIRCLE
13                29 29 8 225 315 ['] Z CIRCLE
14                39 25 35 55 125 ['] Q CIRCLE ;
15
```

```
297 LIST

 0 ( Subroutines Used by MATCH-A1-to-A2                                 )
 1
 2 : B@ j! i! i@ ABS i! j@ ABS j! i@ m@ < NOT IF m@ 2- i! THEN
 3      j@ n@ < NOT IF n@ 2- j! THEN i@ j@ B @1 ;
 4 : AVG-B ['] B dim@ n! m! n@ 0 DO m@ 0 DO I J B@ I 1+ J B@ F+
 5          I 1- J B@ F+ I J 1+ B@ F+ I J 1- B@ F+ 5.0 F/ I J B !2
 6         LOOP LOOP ['] B exc12 ;
 7 ( The following repeats a modified averaging operation on            )
 8 ( matrix B1 by the number of times on the stack.                     )
 9 : REPEAT-AVG-B 0 DO AVG-B LOOP ;
10 ( The following stores sequential numbers from 0 to mn-1 in         )
11 ( matrix B1 starting at {0,0}, going down the column, then          )
12 ( to {0,1} and down etc.                                            )
13 : STORE-INDEX-IN-B2 ['] B dim@ * 0 DO I >N I 0 B !2 LOOP ;
14
15
```

```
298 LIST

 0 : MATCH-A1-to-A2      (       Histogram Matching Routine            )
 1  ['] A ['] B 1 1 ->   ( Transfer A1 to B1.                          )
 2  m@ 2/ REPEAT-AVE-B    ( Repeat averaging operation m/2 times.       )
 3  .01 ['] B a*          ( Divide B1 grey level values by 100.         )
 4  ['] A ['] B A+        ( Add A1 to B1.                               )
 5  ['] A rndA            ( Fill A1 with random numbers.                )
 6  1.0E-12 ['] A a*      ( Divide A1 grey levels by a trillion.        )
 7  ['] A ['] B A+        ( Add A1 to B1.                               )
 8  STORE-INDEX-IN-B2     ( Store sequential number indices in B2.      )
 9  ['] B sortA           ( Sort B1 and B2 with respect to B1.          )
```

```
 10  [']  A exc12          ( Transfer A2 to A1.                          )
 11  [']  A sortA1         ( Sort A1.                                    )
 12  [']  A exc12          ( Transfer A1 to A2.                          )
 13  [']  B [']  A 2 1 ->  ( Transfer indices in B2 to A1.               )
 14  [']  A sortA          ( Sort A1 and A2 with respect to indices      )
 15  [']  A exc12 ;        ( Transfer snow storm from A2 to A1.          )

299 LIST

  0 ( Demonstration of Histogram Matching, 12 hrs. Use PRINT DEMO )
  1                      : ADVANCE 0 DO CR LOOP ;
  2 : DEMO
  3 SMILEY-FACE          ( Generate smiley face in matrix A1.          )
  4 OUTPIC               ( Print smiley face.                          )
  5 17 ADVANCE           ( Advance paper.                              )
  6 [']  A [']  B 1 1 -> ( Transfer smiley face from A1 to B1.         )
  7 [']  A rndA          ( Fill A1 with random numbers.                )
  8 31.5 [']  A a*       ( Adjust grey level values.                   )
  9 OUTPIC               ( Print random image, "snow storm".           )
 10 17 ADVANCE           ( Advance paper.                              )
 11 [']  A exc12         ( Transfer snow storm from A1 to A2.          )
 12 [']  B [']  A 1 1 -> ( Transfer smiley face from B1 to A1.         )
 13 MATCH-A1-to-A2       ( Match histogram.                            )
 14 OUTPIC ;             ( Print smiley face matched to snow storm.    )
 15

300 LIST

  0 (    DEMO uses a batch least squares algorithm to fit a 10th  )
  1 ( order polynomial to 320 scattered data points and then      )
  2 ( displays the data and polynomial graphically, all in        )
  3 ( 5.628 seconds.                                              )
  4
  5 320 12 matrix XX 320 1 matrix Y
  6 12 12 matrix XTX 12 1 matrix XTY 12 1 matrix X 12 1 matrix B
  7 : FUNC SIN RND .5 F* F+ .25 F- ;
  8 ~ : DATA 320 0 DO I >N 50.77042685 F/ FDUP I 0 XX !1 FUNC FDUP
  9            I 0 Y !1 75.0 F* 99.0 F+ N> I PLOT LOOP ;
 10 : REG [']  XX polyA1 [']  XX [']  XX [']  XTX T* [']  XTX INV
 11       [']  XX [']  Y [']  XTY T* [']  XTX [']  XTY [']  B A* ;
 12 : POLY [']  X polyV1 [']  X [']  B V* ;
 13 : DRW 320 0 DO I >N 50.77042685  F/ POLY 75.0 F* 99.0 F+ N> I
 14       PLOT LOOP ;
 15 : DEMO DARK 1 COLOR-SET WHITE DATA REG  DRW ;


301 LIST

  0 (      Batch least squares algorithm is used to find 12th     )
  1 ( order polynomial approximation of the cosine function       )
  2 ( between -PI/2 and PI/2. Time = 16.5 sec. Error = 2.1E-12    )
  3
  4 1001 13 matrix XX   1001 1 matrix Y     variable ERROR
  5 13 13 matrix XTX 13 1 matrix XTY 13 1 matrix X 13 1 matrix B
  6 : REG [']  XX polyA1 [']  XX [']  XX [']  XTX T* [']  XTX INV
  7       [']  XX [']  Y [']  XTY T* [']  XTX [']  XTY [']  B A* ;
```

(Continued on next page)

Fast Matrix Operations (Listing continued, text begins on page 44)

```
 8 : POLY [']  X poJ  '/1 [']  X [']  B V* ;
 9 ~ : DATA 1001 0    I 500 - >N .00105 F* PI F* FDUP I 0 XX !1
10           COS I 0 Y !1 LOOP ;
11 : FIND-COEFICIENTS DATA REG [']  B 0 C1. ;
12 : MAX-ERROR? 0.0 ERROR !1  1001 0 DO I 500 - >N .001 F* PI F*
13              FDUP COS FSWAP POLY F- FABS FDUP ERROR @1 F>
14              IF ERROR !1 ELSE FDROP THEN LOOP
15              ERROR @1 CR CR ." Maximum Absolute Error = " N. ;
```

```
302 LIST

 0 (  Sequential Least Squares Applied to Time Series Analysis   )
 1
 2 (      The next two screens contain a program using a          )
 3 ( sequential least squares algorithm.  To demonstrate          )
 4 ( use n DEMO where n is the number of desired iterations       )
 5 ( divided by 1000 .  For example, 1000 DEMO will iterate       )
 6 ( one million times.  The program will find the autoregressive)
 7 ( coeficients stored in A on line 15 of this screen.  Here A   )
 8 ( is used to generate , Xk, the signal that is being analysed.)
 9 ( This program takes 22.7 seconds per 1000 iterations.         )
10
11
12 5 1 matrix A 5 1 matrix X 5 5 matrix P 5 1 matrix Xr 5 1 matrix
13 PX 1 5 matrix XTP 5 1 matrix B 5 5 matrix PXXTP SVARIABLE N
14 SVARIABLE COUNT
15 .6 0 0 A !1 -.4 1 0 A !1 -.2 2 0 A !1 -.3 3 0 A !1 -.1 4 0 A !1
```

```
303 LIST

 0 ( Sequential Least Squares, Continued                          )
 1
 2 CODE (shiftA1) m LDA n MUL 0 1 MOV adr LDA 65520 # W MOV
 3 BEGIN 0 DS LSG R64 W ) FLD 1 FXCH R64 W ) FSTP 0 INC LOOP 0
 4 N) FSTP 2 2 SUB 2 DS LSG NEXT : shiftA1 dim@ n! m! (shiftA1) ;
 5 : Xk [']  X [']  A V* RND .5 F- F+ FDUP [']  X shiftA1 ;
 6 : Pr [']  P [']  Xr [']  PX A* [']  Xr [']  P [']  XTP T* [']  PX [']
 7 XTP [']  PXXTP A* [']  PX [']  Xr V* 1.0 F+ FNEGATE 1/N [']  PXXTP
 8 a* [']  PXXTP [']  P A+ ;
 9 : Ar [']  P [']  Xr [']  PX A* Xk FDUP [']  Xr [']  B V* F- [']  PX
10      a* [']  PX [']  B A+ [']  Xr shiftA1 ;
11 : DEMO o! [']  B clr [']  Xr clr [']  P I2! [']  P exc12 1.0E+15
12 [']  P a* 0.0 COUNT S! 10.0 N S! o@ 0 DO 1000 0 DO Pr Ar 1.0
13 COUNT S@ F+ FDUP COUNT S! N S@ F= IF CR COUNT S@ 0 F. ." Iterati
14 ons " [']  B 0 C1. N S@ 10.0 F* N S! CR THEN LOOP LOOP ;
15
```

End Listing

And Still More Fifth Generation Computers!

The good Doctor enjoys seeing dialogue in his pages on timely topics. Richard Grigonis originally submitted a guest essay entitled "Fifth Generation Computers" in the December 1982 issue of DDJ, *in which he briefly presented his vision of microcomputers in the 1990s. Those predictions were challenged in the March 1983 issue. In response, Mr. Grigonis presents here a more detailed treatment of the ideas from the original essay, as well as some further predictions and his conceptualization of a fifth-generation workstation.* —*Ed.*

Open Forum's "Comments on Fifth Generation Computers," (March 1983 *DDJ*, No. 77) by Michael J. Doherty, contained some perceptive insights challenging the fifth-generation supermicro that I proposed in "Fifth Generation Computers" (December 1982 *DDJ*, No. 74). He was particularly critical of the preciseness of my predictions regarding what he calls the "Grigonis 64/110 processor," which he seems to think would violate the known laws of physics.

This article, however, shall serve to nullify Mr. Doherty's arguments and also to satisfy his curiosity to see, in his words, "Grigonis' speculations on the workstation environments that will have to be developed to contain the operator as well as the advanced hardware he describes." Moreover, in this article I will do both him and the readers of *DDJ* a service by precisely predicting the development of computers (and their workstation environments) over the next 50 years.

First, let me recount the features of the fifth-generation supermicro, expanding my explanations as we go along.

64-bit processor (von Neumann architecture) running at 110 MHz.

Doherty thought that putting the equivalent of a Cray 2S on a chip was impossible. He was thinking specifically of certain physical boundary conditions imposed by quantum mechanics – namely, the Planck length from which is derived the Heisenberg Uncertainty Principle, in this case applied to the probable locations of the electrons in the signal pathways of the processor that would set up unusual field effects in neighboring signal pathways. Doherty also thought that too many gates would be required and that the switching times would not be fast enough to handle the signal densities required for a supercomputer.

I am sure that the above would come as a surprise to the scientists at the Lawrence Livermore Laboratory who plan to develop the Mark V, a microprocessor four times as fast as a Cray!

Let's also take a look at the Department of Defense (DoD) and their Very High-Speed Integrated Circuits (VHSIC) Program. Phase Ia of this program is to develop chips with a 1.25 micrometer minimum feature size and an equivalent gate-clock frequency product exceeding 5×10^{11} gate-Hz/cm^2 which would have a maximum power dissipation not exceeding 3 watts/cm^2. The minimum clock speed goal is 25 MHz.

Phase IIb will achieve clock rates above 30 MHz; processor signal pathways will have submicrometer feature sizes (0.5 to 0.8 micrometer) and higher gate densities; and the gate-clock frequency product will increase to about 10^{13} gate-Hz/cm^2.

The manufacture of such processors goes beyond present IC fabrication techniques, since optical lithography is bound by a one-micrometer "barrier" imposed by the highest frequencies of visible light, and will require a new technology. Phase III, therefore, will examine the use of "carving" out tiny processor signal pathways and electron beams (e-beams) or using lithographic techniques at electromagnetic frequencies higher than that of visible light – namely, X-rays.

Additional speed will be obtained simply by rethinking the whole architecture of processor design with the aid of some Computer Aided Design (CAD) such as McWilliams' Structured Computer Aided Logic Design (SCALD) program, and Artificial Intelligence programs. We will probably end up with optical computer chips based on matrix-processing techniques by the end of the next decade, quickly supplanting the gallium arsenide (GaAs) chips that will be just beginning to flourish in the early 1990s. Besides, what's so great about using GaAs as an IC substrate anyway? Its much-touted 7-to-10-fold speed advantage over silicon is dwarfed by silicon's 100-fold gate density advantage.

Interestingly, the speed of light dictates that all of the most advanced mainframes appearing at the end of the next decade will in fact be either micros or at least the *size* of micros. If the travel time of the signals between switches does not exceed the switching time, then the distance between the farthest two switches, or the length of the longest signal pathway, must always be less than or equal to the switching time multiplied by the speed of light. As Seymour Cray learned, if the switching time is one nanosecond (10^{-9} sec), then the length of any signal pathway in the processor must not exceed 30 centimeters. Mainframe processors at the beginning of the next decade will have switching times of about 10^{-10} of a second, so their signal pathways cannot exceed 3 centimeters. Computers around the year 2010 will have switching times of about 6 *picoseconds* (6×10^{-12} sec), restricting signal pathway lengths to a little over 0.15 centimeters! Quite a big "mainframe," eh?

Josephson circuits can already be switched in 13 picoseconds, 7 picoseconds for the actual switching time and 6 picoseconds for the signal to traverse the circuits. This is approaching as fast a switching time as is theoretically allowed. And since the signals travel at nearly the speed of light in the Josephson circuit substrate material (0.003 inch per picosecond – and that's with a minimum line width as large as 2.5 microns), the problem now is not to make the switches faster, but to place them closer together.

Actually, Doherty is sort of right, but the Uncertainty Principle enters into the discussion in a different way. The fundamental limit of all computation, even that of the human brain, is mathematically *identical* with the Uncertainty Principle of quantum mechanics. By this, I mean that the acts of signals triggering switches are physical measurements, and the accuracy of all physical measurements is subject to the Uncertainty Principle. The faster the measurements, the more unreliable they will be, resulting in a downward spiral of more and more error-correcting circuitry.

We can reduce error by increasing the signal energy, but we cannot continue doing this indefinitely, as the processor will melt! According to physicist Hans J. Bremermann, the "trade-off" between the number of distinguishable signals (information) that can be sent is proportional to the time required to identify

by Richard Grigonis

Richard Grigonis, Children's Television Workshop, One Lincoln Plaza, New York, New York 10023.

them: $E/h[sec^{-1}]$, where E is the energy available for signalling, and h is Planck's constant. This limit applies to the human brain as well as computers.

Another physical limit, also discovered by Bremermann in 1961, takes into account the fact that the energy of the signals has a "mass" just like matter: no computer can have a signal flow exceeding mc^2/h bits per second, where m represents the total mass of the system, c is the velocity of light, and h is Planck's constant. Theoretically, the maximum signal flow for *any* calculating device or even *any* physical implementation of *any* algorithm is 1.35×10^{47} bits per second per gram of system weight. Since today's supercomputers run at about 10^5 bits per second per gram, we have a long ways to go before hitting the maximum.

Yet a third limit is not a physical limit, but a limit of algorithm complexity. This is not quite as bad as the other two limits since we can always find shortcuts to mathematical problems once considered hopeless, or "transcomputable." A perfect example of this is the travelling salesman problem. A salesman must find the shortest path between, say, 10 cities. The trial-and-error computational cost of doing this is 181,440 steps. One can see that this problem is important to hardware designers as they are always figuring out ways of cramming the most circuits in the least volume by finding the shortest signal path between switches. The difficulty comes when we increase the number of cities or switches to 100, which requires 10^{100} steps. Calculating the distance between 400 cities would require 10^{800} steps. This figure vastly exceeds the 10^{120} total possible moves in chess!

But now three IBM researchers have discovered that some equations from statistical mechanics provide an amazing shortcut to the problem. If we pretend that the circuits are water molecules, then the shortest length through all of them can be found if we "cool" them, bringing them closer together. The minimum distance between 400 circuits can now be calculated with just a million (10^6) steps!

I think Mr. Doherty, therefore, should not worry about "theoretical limits."

An actor-based program and language generator. A sort of Smalltalk version of The Last One which develops the optimum language *for execution as well as the code defining the flow-of-control.*

Now that we are on the subject of languages, I must mention the Japanese Fifth Generation project, which started this whole debate. Let me state that although the Japanese will undoubtedly develop the tremendously advanced hardware they have set as a goal for themselves – a computer 100 times faster than a Cray 1 – the fatal flaw in the project is the software, particularly the use of the language Prolog in their Artificial Intelligence programs.

The first problem with the Japanese is that they have decided to take a creative lead in the area of supercomputer hardware and software. Under no circumstances, therefore, are they going to imitate what the Americans have been doing with Artificial Intelligence. The initial result of this policy was their decision not to use LISP, the *lingua franca* of American AI research.

I have a few bones to pick with LISP myself: the parentheses notation, the fact that it is slow (except for the optimized MACLISP math functions) both because the language is interpreted and because the processor must chase thousands of pointers around, the unsophisticated dynamic scoping, the method of garbage collection, its near non-transportability (as opposed to Ada or "C"), and the ominous fact that although it is the second oldest computer language (McCarthy started working on it in 1958, a year after the first Fortran compiler appeared) it has never been used for anything outside the field of Artificial Intelligence!

Still, LISP can treat its own programs as data, can modify itself or write new programs *and* evaluate them. The language is extensible, and the eight or so basic, primitive functions constitute a universal Turing machine and can be used to calculate anything that can be calculated. LISP has been around so long that there exist sophisticated software/hardware development tools for it such as the Xerox and Symbolics 3600 "LISP Machines."

So what have the Japanese selected instead? Prolog, an obscure language developed by the French and "polished" a bit by the British. Prolog is the fundamental error in the otherwise sound Japanese Fifth Generation project.

This brings us to the second problem with the Japanese, which is their quest for absolute perfection. This seems to be an outgrowth of Japanese philosophy in general, and had until recently motivated them technologically to make fools out of their American competition. One of the reasons they chose the obscure Prolog language was their problems with understanding traditional Western programming languages. The Japanese and Chinese use ideographic, "picture" languages to communicate with each other instead of the linear, lexical, "letter-and-word" languages of the Europeans. The Japanese thus tend to think in terms of patterned wholes, whereas Westerners think along linear, reductionist sequences. This leads to interesting effects: Someone in Japan can look at the pictograms in a Chinese newspaper and understand the meaning of the words and the sentences without having the slightest idea of their pronunciation, while an American can look at an Italian newspapers pronounce the words correctly, even identify the nouns and verbs correctly, *but cannot understand a thing!*

The one area where the Japanese computer scientists are on an equal footing with their American counterparts is in a different linguistic realm entirely – namely, the notations of pure symbolic logic, as developed by Boole and Frege. What attracted them to Prolog is that its "horn-clause" notation and axiom-theorem structure force the programmer to use the first-order predicate calculus. Thus, instead of millions of *arithmetic* calculations per second, the Japanese will measure the power of their computer in terms of the number of what they call "LIPS" or Logical Instructions Per Second. Now, although representing knowledge *procedurally* via heuristics in LISP is a standard way of doing things in American AI programs, the only *declarativist* means of representing knowledge that ideally surpasses procedural representations is *logic*. Hence, Prolog became the language of choice for the Japanese.

I mentioned above that logic is ideally a *perfect* way of representing knowledge – but the qualifying word is "ideally." In reality, logical knowledge representation and inference calculation is a nightmare for a computer. The addition of just a few hundred facts to the "knowledge base" forces the computer to calculate *all* of the possible logical relationships (inferences) between all of the other facts currently stored in the system and the new facts themselves, resulting in a "combinatorics explosion" that fills up both the internal memory and the disk storage. Even a slight change or revision of a few crucial facts can cause an avalanche of updating.

Various ways have been tried to get around this problem. Some Stanford researchers are attempting to add only temporary "local" changes to the knowledge base, recalculating all of the logical relationship changes only periodically. Apparently, the Japanese plan to conquer the problem with brute force, building computers with huge amounts of internal memory, disk storage, and a processor that may achieve billions of logical operations in a second.

But before their grand supercomputer is ready, the Japanese are planning to market (sometime in 1984) smaller, personal computer versions of their system, called "Prolog machines." The Japanese seem aware of the problems they have created for themselves and are not entirely excluding the possibility of using LISP functions. Indeed, one of the companies participating in the project has a goal of producing the world's fastest LISP machines as well. The Japanese insist that they are "spelling prolog with a small *p*," meaning that they may not be using just the strict logical structure of Prolog (with

a capital *P*) as originally developed by the Europeans.

4-megabyte EEPROM memory chips, with a total of 32 megabytes of memory.

10,000-to-15,000-megabyte optical disc *drives. (The industry, when describing optical storage, spells disk with a "c"; when referring to magnetic storage, disk is spelled with a "k.")*

Flat screens displaying three WP pages, or three sections of a DP program simultaneously, in color.

Rather than a single wide screen, in the manner of the inexpensive, portable-*looking* model that artist, Al McCahon, drew to illustrate the cover of the December 1983 *DDJ* issue, the office version will actually be constructed so that the three main screens are arranged in a triptych, reminiscent of the arrangement of screens employed in the recent release of the movie *Napolean*. In this way, both the optical and geometric centers of the screen are kept equidistant from the eyes of the viewer, so the user can look at any screen in succession without eyestrain.

Also, beneath the corners of both the left and right screens will be located two additional, smaller screens. One screen will display the page appearing *prior* to the one currently displayed on the larger screen above, the other one displaying the page *after* it. While working on one of the peripheral larger screens then, the user need only glance down to look at preceding or succeeding pages, if any, instead of swinging one's head 150 degrees to the farthest of the larger screens. In this way the user can command the three large screens to hold three pages from totally different sections of a file undergoing editing without the fear of losing one's perspective of the continuity of the file. The central screen does not have subsidiary screens, partly because of the space taken up by the keyboard and partly because users will probably use the central screen as a sketchpad, or a temporary work area (such as the disembodied "work page" as can be currently found in the WP documents on Wang equipment) where material from various areas is stored and polished before it is reinserted in the file proper.

A mouse replacing the cursor directional keys. This prediction came true a lot sooner than 1992 (it was the month after the appearance of my article, in fact), when the one-button mouse was announced as standard equipment on the new Apple Lisa.

By the turn of the century, however, the mouse will have a successor – two successors, actually. One is an attachment either built into the headrest of the user's chair or else set on each of the three screens, consisting of a low-power semiconductor laser projecting and scanning an invisible light beam (in a harmless end of the spectrum) onto the user's eyes. A flat semiconductor "lensless" camera will read the light of the reflected beam and track the movements of the user's eyes across the screen. If the user wishes to make a specific selection on a menu, one will focus one's gaze on the appropriate icon and press a foot pedal, telling the computer that the current line of sight is to be construed as a menu selection or perhaps an INPUT command if the action occurs during a program run.

The above is fine for repositioning the cursor quickly *within* the boundaries of the screen. But what about scrolling the screen "windows" through a file? This is where the second device comes in – a box attachment to the armrest upon which is mounted an arcade-style "trackball" and some buttons. In the WP editor, spinning the ball to the right, for example, will "flip" pages from the small right-hand corner screens (succeeding pages) to the larger screens, and then back again to the smaller, left-hand screens (preceding pages), until the end of the document is reached. It should be amusing to set up all the screens so that they are looking at one segment of seven continuous pages and then spinning the ball and watching a train of pages appear in succession on all of the screens (sort of like doing word processing in Cinerama!). When the user is working on a DP spread-sheet, the trackball can be used to scroll either sideways across the matrix, up-and-down, or even on a diagonal.

When the screen stabilizes, the eye scanner cursor controls will again become functional. If the user is not looking at an icon, but at a line from a WP or DP file when he or she presses the pedal, then the computer considers this action to be an EXECUTE command in the DP/WP editor. In this way, editing would still consist of the user pressing INSERT, DELETE, REPLACE, or MOVE keys, but doing so would cause the laser scanning device to activate the chosen editing command at the cursor position set by the user's line-of-sight.

A description of a MOVE editing operation in a futuristic operating manual of the middle of the next decade may read something like this:

(1) Deactivate the screen cursor by looking at the beginning of the block of characters to be moved and pressing the MOVE button. The first character of the block that you are looking at should now be highlighted. Do not press any alpha-numeric character keys at this point, as it will cancel the operation and the normal screen cursor will reappear, but this time at the screen location at which you were just looking.

(2) Continue highlighting the block to be moved by reading the text until you reach the end of the block or by just glancing down at a specific character (period, paragraph return character, etc.). Remember, the text will be highlighted from the initial point at which you pressed the button to the point you are currently looking at. Looking to the left or up on the screen will dehighlight the formerly highlighted areas.

(3) When you are satisfied with the size of the highlighted block to be moved, keep staring at the last character of the block and step on the ENTER/EXECUTE pedal. The highlighted area will disappear off of the screen.

(4) Reposition your gaze at the new location where the block is to be inserted. If you must give commands to go to a totally different file, then the system will ignore your line-of-sight gaze until it is again within the stabilized screen area. Look at the screen location where the block is to be inserted.

(5) Step on the ENTER/EXECUTE pedal. The block of characters should now have been inserted starting at the last screen location you were looking at when you pressed the pedal. Notice that the cursor has now reappeared, also in this same location at the beginning of the moved block.

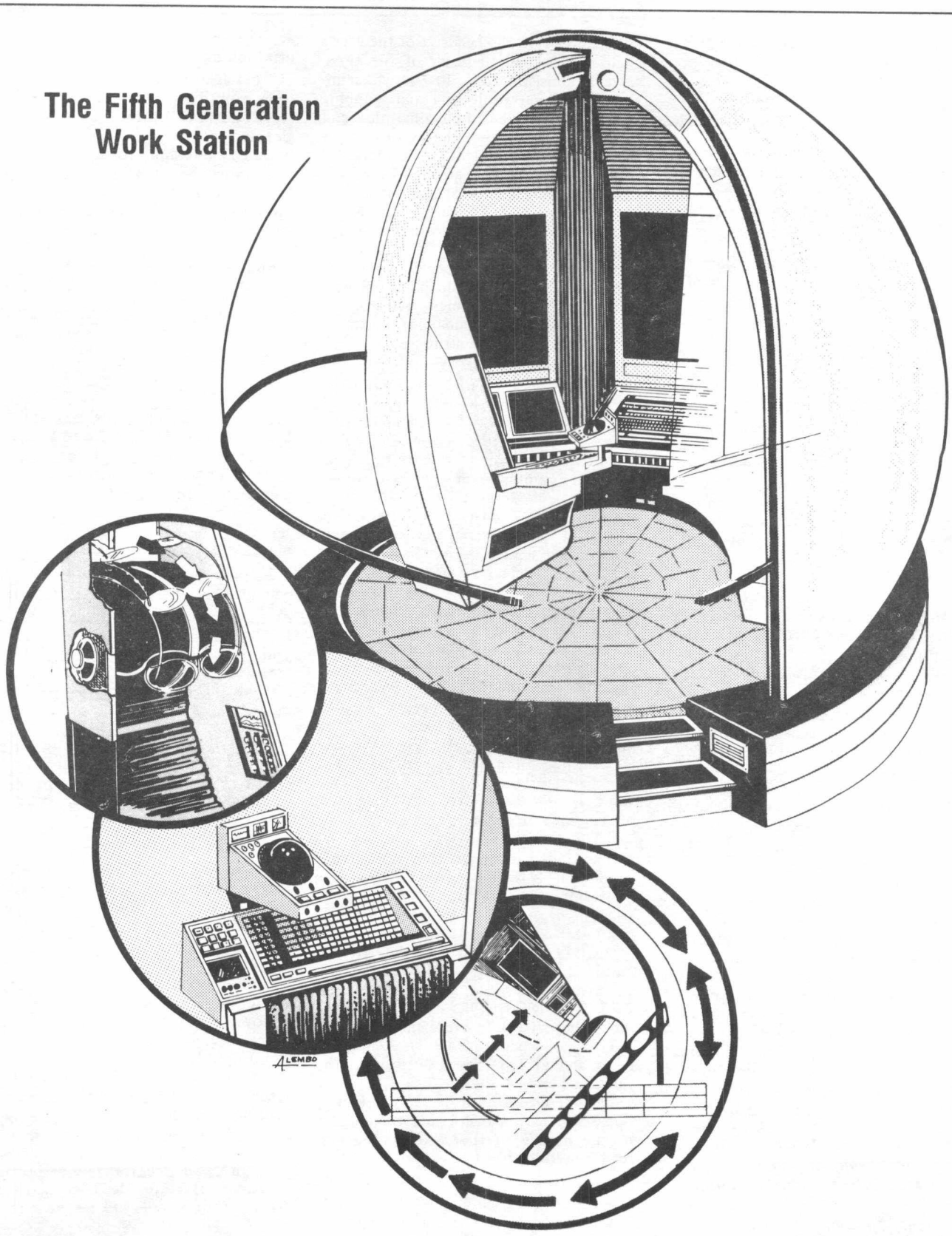
There may be a few other features, such as a fully programmable keyboard that allows the user to reconfigure the keys to the Dvorak "speedboard" arrangement.

The Workstation

The main emphasis in the design of the workstation – and the general working environments – of the 1990s and beyond will be a total dedication to efficiency and productivity, regardless of initial start-up costs or peculiar design features. We will have come far from the days of World War II, when defense contractors discovered that productivity could be improved by putting back supports on the stools used by assembly line workers!

The offices of the future will be highly security-oriented. There will be no harsh lighting, and the rooms containing the supermicros and their workstations will be set in a dust-free environment – no dust, dirt, pollen, extraneous carbon monoxide, or sufur dioxide from the exterior urban environment will be allowed into the office. Negative ions and additional oxygen will be pumped into the air. Many employees will be working at home, however, via telecommuting.

The office visitor will first notice the workstations: a number of large, off-white plastic arches, looking a little like



Conceptual drawing of the Fifth Generation Workstation as conceived by author Richard Grigonis. Illustrated by A. Lembo, an artist from Bell Laboratories.

miniature quonset huts, each sitting atop a short pedestal. A series of two or three steps leads up the side of each pedestal to the arch. These workstations are scattered around the room, with large numbers affixed to the back of them.

Looking inside one of the workstations, one sees our old friend the supermicro, the keyboard/screens/chair framework set within the arch. Modular construction seems evident and whole sections (keyboard, screens) can be replaced easily. The user climbs into the chair and lowers what appears to be two large additional armrests that pivot down on either side, each actually a sort of counter, which corral the user in the workstation. Florescent-type bulbs built into the counter backboards illuminate any odds and ends the user wishes to place on the counters to refer to in the course of a session with the computer. One notices that all the interior plastic is of a soft, non-glare, earth color variety.

The pressure of sitting in the chair automatically activates the low-intensity flat screens. Normal room light would wash them out, which is another reason for the totally enclosed workstation. Another reason is that the user has privacy and has the option of turning on stereo loudspeakers built into the chair headrest that can play music, "white noise," nature sounds, etc.

The user sees some drawers set beneath the keyboard. Instead of keys, authorized personnel are able to open drawers (and similarly gain access to certain files and functions) through voice recognition. The user will probably sit in the chair and deliver a rambling monologue into a microphone built into the workstation until the computer signals that it has identified the user and has given him or her the authority to use the designated functions and files.

The chair/screen/keyboard motor controls can be found on the chair armrests. The user moves the chair a bit closer to the screen and then, deciding to work in a more supine position, presses another button and rotates the framework backwards. As the chair leans back, the keyboard moves up and stays in the same relative position the user had set it in earlier, as do the screens, following tracks built into the underside of the arch.

After selecting some "environmental sounds" to play through the headrest speakers, adjusting the temperature of the chair, and setting the negative ion level in the workstation atmosphere, the user is off and running . . . .

2000 and Beyond

Since *DDJ* is essentially a systems software journal, I will only briefly describe sociological developments occurring as a consequence of this kind of equipment beyond the year 2000. Besides, it is best that the public not know that they, within 50 years, are all going to resemble hackers to one degree or another. In any event, I predict the following:

Mass transit is a mass waste of time, so telecommuting will flourish. There will still be work centers, but part of one's salary will no longer be devoted to transporting oneself to the office.

For nationwide teleshopping, the user at home receives a holographic plate in the mail upon which is stored the contents of, say, a Sears catalogue. Many separate holographic images can be stored on one photographic plate simply by changing the angle of the reference beam between exposures. The user puts the plate in a viewing device and spins it to change the direction of the illuminating beam and observe the different products. Selecting one, the user calls up his videotex system and finds out which "point of distribution" (a cross between a store and a warehouse) has the best price. After signalling, the transaction is recorded and an extensive delivery system goes into action, bringing the product to the user's home.

The streamlining of oral and written language, with so many people using computers, means the written language will take on a more ideographic quality. We will see the appearance of "videoglyphs" (as some researchers call them), which are ways of throwing entire concepts and huge blocks of information on a screen very quickly in pictographic form. This will damage the appreciation of our slower, more flowery written language even more than did television. Our written and oral language will seem cumbersome.

Since computers will give their users nearly instantaneous responses, people will communicate with the equipment at their own pace. After years of doing this for each working day (or school day, in the case of children), catastrophic effects on interpersonal communication will result. People will suddenly lose the ability to tolerate others not spewing out or receiving information as fast as they do in a conversation. I see this phenomenon appearing among hackers right now. The solution to this interpersonal communications problem seems to be the one the hackers have already, albeit unconsciously devised: Although two of them are sitting at terminals near each other, they communicate via the computer with themselves and as many others in their telecommunications network as they can comfortably accommodate. We will live in a world where everyone will be an acquaintance of everyone else, but few personal friendships will exist.

Also, since all of the world's information will eventually be in the form of huge databases, advanced information retrieval techniques of document titles alone (like those employed by the Lockheed Dialog system) will prove to be insufficient and will overwhelm the user. Thus, an Artificial Intelligence program – a natural language processing front end, to be specific – will have to be placed between the user and the enormous databases.

The users will now be able to access any information they please at the push of a button. Society will reason that a generalized liberal arts education is too expensive and, in any case, is no longer necessary since the information is always there if one has to look it up. The problem is – nobody will look it up. The educational curriculum will become highly specialized. This, plus the long-term exposure to supercomputer AI programming, will transform the users into what might best be described as idiot savants, or hackers without the technical expertise.

Entertainment will be provided by huge arcades where three-dimensional "entertainment experiences" will occur. These experiences will be far more "intensive" than what can be conceived of at present since people become bored very easily with any particular level of sight-and-sound effects technology and crave something new. One realizes how trite Space Invaders-type games appear to us now, as opposed to how exciting they seemed just a few years ago.

Schools will be replaced with "educational interaction centers." The less said about them the better. After all, imagine your child sitting in a dark room at a long row of keyboards, being "programmed" by society to become an extension of the equipment, not vice versa.

Get the picture? DDJ

RED
A Better C Screen Editor, Part II

Last month I introduced RED, a new screen editor which is written in Small-C. That article covered RED's buffering scheme in great detail. This month I'll discuss the Small-C Library used in RED and improvements that might be made to RED. I'll also say a few words about why RED is copyrighted.

To the user, RED appears very similar to the ED2 editor described in the January 1982 issue of *Dr. Dobb's Journal*. Because of this, the remainder of RED is provided in Listing One (page 66) without extensive discussion. Tables 1, 2 (at right), and 3 (page 64) give a brief summary of RED for those not familiar with the original editor.

A New Run-Time Library for Small-C

The buffer routines described last month require a run-time library which supports "unbuffered," i.e., block-at-a-time, file access. The new library presented in Listing Two (page 87) is an edited and slightly modified version of part of the BDS C run-time library. (There is a lot more to the library that I have not included.) This library is presented here by permission of its author, Leor Zolman.

I have tried to make sure that the routines presented here work *exactly* the same as the original BDS C library routines. Changes to the code were made only where absolutely necessary and are marked in the listing. The changes were made because Small-C pushes arguments on the stack in reverse of the order that BDS C uses. Small-C pushes parameters in the order in which they appear in the parameter list. Thus, the last argument in the list appears at the *top* of the stack (right under the return address). In BDS C, the *first* argument appears at the top of the stack.

The new library passes command line arguments to the main() function. To retrieve these arguments, add two arguments to main() as follows:

by Edward K. Ream

Edward K. Ream, 1850 Summit Avenue, Madison, Wisconsin 53705.

Copyright © 1983 by Edward K. Ream. RED may be used for non-commercial purposes only. No commercial use of RED may be made without the author's expressed written permission.

Table 1
COMMAND MODE

append *file* . . . . . .Append *file* after the current line.
change *line range* . . . . . .Change lines which match a pattern.
? in search mask matches any character.
? in change mask becomes character that matched corresponding ? in the search mask.
^ in search mask in column 1 matches the start of a line.
clear . . . . . .Clear the buffer.
copy *line range number* . .Copy lines in *line range* after line *number*.
delete *line range* . . . . . .Delete ALL lines in *line range*.
dos . . . . . .Exit from RED.
find . . . . . .Go to next line that matches a pattern.
? and ^ as in change command.
g *number* . . . . . .Go to line *number*.
help . . . . . .Print a help message for command mode.
list *line range* . . . . . .List lines to the printer.
load *file* . . . . . .Load the buffer from a file.
move *line range number* . .Move lines in *line range* after line *number*.
name *filename* . . . . . .Make *filename* the current file name.
resave . . . . . .Save the buffer to an existing file.
save . . . . . .Save the buffer to a new file.
search *line range* . . . . . .Search for a pattern.
? and ^ as in change command.
tabs *number* . . . . . .Set tab stops to every *number* column.

Table 2
SPECIAL CHARACTERS

Key	Default Value	Result
up key:	control-u . .	Move the cursor up. Enter edit mode.
down key:	control-d . .	Move the cursor down. Enter edit mode.
right key:	control-r . . .	Move the cursor right.
left key:	back space . .	Move the cursor left.
insert up key:	LF	Insert a new line above the current line. Enter insert mode.
insert down key:	CR	Insert a new line below the current line. Enter insert mode.
delete character key:	DEL	Delete one character.
delete line key:	control-z . .	Delete the current line.
insert key:	control-n . .	Enter insert mode.
command key:	escape	Enter command mode.
edit key:	control-e . .	Enter edit mode.
undo key:	control-x . .	Undo changes to the current line.
split key:	control-s . . .	Split the current line.
join key:	control-p . .	Join current line, preceding line.
repeat key:	conrol-a . . .	Repeat the previous character.

```
main(argc, argv)
int argc;
int * argv;
  /* Kludge: should be char ** argv */
```

Argc is *one more than* the number of arguments on the command line. Argv is a pointer to an array of pointers to the actual arguments. Argv [0] is not used, thus argv [1] is a pointer to the first argument. Because of the limitations in Small-C, argv must be declared as shown.

The library also provides the following unbuffered I/O primitives. These *primitives* work just the same as the primitives in the BDS C Library.

close (fd)
int fd;

Closes a file which was opened by open() or creat(). Returns -1 on an error.

creat (filename)
char * filename;

Creates an empty file. The file is erased if it exists. The file is opened for reading or writing. Returns a file descriptor (a short integer) which is used by the read(), write(), seek(), tell(), close(), and fabort() routines.

fabort (fd)
int fd;

Frees the file descriptor fd without closing the file. Don't use the fabort() on a file which was open for writing unless you are willing to lose data which were written into it.

open (filename, mode)
char * filename;
int mode;

Opens a file for reading (mode 0), writing (mode 1), or both reading and writing (mode 2). The file must already exist; -1 is returned if it doesn't. Returns a file descriptor on a successful open.

read (fd, buffer, nbl)
int fd;
char * buffer;
int nbl;

Reads nbl sectors of the file whose file descriptor is fd into the buffer. All sectors are 128 bytes in the CP/M world, regardless of their actual size on your disk. Returns the number of blocks actually read, or -1 for errors. 0 means end of file.

rename (oldname, newname)
char * oldname, * newname;

Renames the file.

seek (fd, n, code)
int fd, n, code;

Positions the file for reading or writing at a particular sector. Positions the file at the nth sector if code is 0. Positions the file n sectors past the present sector if code is 1. Returns -1 on a seek for a non-existent sector. Make sure to close the file if seek fails; the file is sure to be in bad shape.

tell (fd)
int fd;

Returns the current position of the file (an integer). The first sector of the file is sector 0.

write (fd, buffer, nbl)
int fd;
char * buffer;
int nbl;

Writes nbl sectors from the buffer to the file. Returns -1 on an error, which is usually caused by the disk becoming full.

unlink (filename)
char * filename;

Erases the file (permanently!) from the disk. Use with caution.

Finally, the new library provides two functions which allow jumps between procedures. This is a very important capability to have for error recovery.

setjmp (buffer)
char buffer [6];

longjmp (buffer, val)
char buffer [6];
int val;

Setjmp() saves the current state of the processor in the buffer. (For Small-C this means the BC register, the program counter, and the stack pointer.) Setjmp() itself always returns 0, but see below. Whenever a later call to longjmp() is made (from *anywhere* in the current function or a lower-level function) the CPU state is restored to what it was when setjmp() was called last with the same buffer argument. The program behaves as if control were returning from the setjmp() function *except* that the return value is the "val" which was passed to longjmp().

Ideas for Improving the Editor

Nothing is impossible for the man who doesn't have to do it himself.
– Weiler's Law

The algorithms presented last month are the best that I know of; I would like to hear about any improvements that you might have. Of course, improvements to the editor are possible. Here is a short list.

Rewriting functions in assembly language is an obvious way to increase speed by 10% to 100%. The swap_in(), b_scan(), and bufgetln() routines would be my candidates for recoding because they are used so often. It is likely that other routines would also benefit from being written in assembly language.

Another idea is to change the format of the data area of a block. Instead of preceding each line with a count, it is possible to create a table of pointers to the start of each line. This table would make searching in the b_scan() routine unnecessary. Dave Cortesi discussed this idea and others at length in Dr. Dobb's Clinic (*DDJ* No. 65, March 1982, pages 6-9). Putting such a table in each block will slow down the insert and delete operations because the table must be adjusted. However, that penalty is slight and the gain to be had from speeding up b_scan() is much larger.

Perhaps the most interesting improvement to the editor would be to add multiple windows to the screen. The new buffer routines could be modified easily to deal with multiple files. The idea is to have all files share the same set of buffers; all that must be done to change from one file to another is to swap out all the dirty blocks. Of course, such a change would require a major revision to the window module on file red4.sc.

RED and Copyright

I think it is important to distribute source code for virtually all programs. Just as I used the code in the "Just Like Mom's" editor as a starting point, I hope some of you will take these routines and use them in new ways.

RED is copyrighted solely to protect myself against those who distribute RED without permission. I hope you will feel free to do anything with this editor except distribute it for profit. Please don't let the copyright prevent you from using the code; if you are unsure about how I may react to your plans, just give me a call. I'm sure we can work something out.

Acknowledgements

RED owes much to my friends at the Waisman Center of the University of Wisconsin, Madison; Cliff Gilman greatly improved RED's documentation, Bruce Orchard assisted in moving RED to the Harris computer, and Dave Wilson suggested subtle, but important, improvements in the workings of various commands. DDJ

(Listing One begins on page 66)

(Listing Two begins on page 87)

RED - Listing (Text begins on page 62)

Listing One

```
/*
        RED main program -- small-C version

        Source:  red2.sc
        Version: January 22, 1983

        Copyright (C) 1983 by Edward K. Ream
*/


/* Define the disk recovery point. */

char D_ERROR [6];


/* the main program dispatches the routines that
 * handle the various modes.
 */

#define CMNDMODE 1      /* enter command mode flag */
#define INSMODE  2      /* enter insert modes flag */
#define EDITMODE 3      /* enter edit mode flag */
#define EXITMODE 4      /* exit editor flag */

main()
{

        int mode;

        /* ready the system module */
        sysinit();
        /* clear the main buffer */
        bufnew();
        /* fmt output by default goes to screen */
        fmtassn(NO);
        /* set tabs, clear the screen and sign on */
        fmtset(8);
        outclr();
        outxy(0,SCRNL1);
        message(SIGNON);
        message(VERSION);
        message(COPYRIGHT);
        message("");
        message(XSIGN);
        message(XSIGN1);
        outxy(0,1);
        /* clear filename [] for save(), resave() */
        pmtclr();
        /* start off in command mode */
        mode=CMNDMODE;


        /* get null line 1 for edit() */
        edgetln();
        while(1){
                if (mode == EXITMODE) {
                        break;
                }
                else if (mode == CMNDMODE) {
                        mode=command();
                }
                else if (mode == EDITMODE) {
                        mode=edit();
                }
                else if (mode == INSMODE) {
                        mode=insert();
                }
                else {
                        syserr("main: no mode");

                        mode=EDITMODE;
                }
        }
}

/*
 * handle edit mode.
 * dispatch the proper routine based on one-character commands.
 */

edit()
{
char buffer [SCRNW1];
int v;
int x,y, topline;
char c;
        /* we can't do edgetln() or edgo() here because
         * those calls reset the cursor.
         */

        /* Set disk error recovery point. */
        setjmp(D_ERROR);

        pmtedit();
        while(1){
                /* get command */
                c=tolower(syscin());

                /* comment out ----- 9/14/82
                if (c == 3) {
                        bufdump();
                }
                ----- end comment out */

                if ( (c == ESC1) | (c=='c') ) {
                        /* enter command mode. */
                        return CMNDMODE;
                }
                else if ( (c == INS1) | (c=='i') ) {
                        /* enter insert mode */
                        return INSMODE;
                }
                else if (special(c) == YES) {
                        if ( (c == UP1) | (c == DOWN1) ) {
                                return INSMODE;
                        }
                        else {
                                continue;
                        }
                }
                else if (control(c) == YES) {
                        continue;
                }
                else if (c == ' ') {
                        edright();
                        pmtcol();
                }
                else if (c == 'b') {
                        edbegin();
                        pmtcol();
                }
                else if (c == 'd') {
                        /* scroll down */
                        pmtmode("edit: scroll");
                        syswait();
                        while (bufnrbot() == NO) {

                                if (chkkey() == YES) {
                                        break;
                                }
                                eddn();
                        }
                        pmtedit();
                }
                else if (c == 'e') {
                        edend();
                        pmtcol();
                }
                else if (c == 'g') {
                        /* save x,y in case don't get number */
                        x=outgetx();
                        y=outgety();
                        pmtcmnd("edit: goto: ",buffer);
                        if(number(buffer,&v)) {
                                edgo(v,0);
                        }
                        else {
                                outxy(x,y);
                        }
                        pmtedit();
                }

                else if (c == 'h') {
                        /* remember how screen was drawn */
                        x=outgetx();
                        y=outgety();
                        topline=bufln()-y+1;

                        /* output the help message */
                        outclr();
                        outxy(0,SCRNL1);
                        edithelp();

                        /* redraw the screen */
                        bufout(topline,1,SCRNL1);
                        outxy(x,y);
                        pmtedit();
                }

                else if (c == 'k') {
                        pmtmode("edit: kill");
                        c=syscin();
                        if ( (special(c) == NO) &
                             (control(c) == NO)
                           ) {
                                edkill(c);
                        }
                        pmtedit();
                }
                else if (c == 's') {
                        pmtmode("edit: search");
                        c=syscin();
                        if ( (special(c) == NO) &
                             (control(c) == NO)
                           ) {
                                edsrch(c);
                        }
                        pmtedit();
                }
                else if (c == 'u') {
                        /* scroll up */
                        pmtmode("edit: scroll");
                        syswait();

                        while (bufattop() == NO) {
                                if (chkkey() == YES) {
                                        break;
                                }
                                edup();
                        }
                        pmtedit();
                }
                else if (c == 'x') {
                        pmtmode("edit: eXchange");
                        c=syscin();
                        if ( (special(c) == NO) &
                             (control(c) == NO)
                           ) {
                                edchng(c);
```

```
                        }
                        pmtedit();
                }
                /* do nothing if command not found */
        }
}

/* insert mode.
 * in this mode the UP1, UP2 keys reverse their roles,
 * as do the DOWN1, and DOWN2 keys.
 */

insert()
{
char c;
        /* Set disk error recovery point. */
        setjmp(D_ERROR);

        pmtmode("insert");

        while (1) {
                /* get command */
                c=syscin();
                if (c == ESC1) {
                        /* enter command mode */
                        return CMNDMODE;
                }
                else if (c == EDIT1) {
                        /* enter edit mode */
                        return EDITMODE;
                }
                else if (c == INS1) {
                        /* do nothing */
                        ;
                }
                else if (special(c) == YES) {
                        if ( (c == UP2) | (c == DOWN2) ) {
                                return EDITMODE;
                        }
                        else {
                                continue;
                        }
                }
                else if (control(c) == YES) {
                        /* ignore non-special control chars */
                        continue;
                }
                else {
                        /* insert one char in line */
                        edins(c);
                        pmtcol();



                }
        }
}

/* return YES if c is a control char */

control(c) char c;
{
        if (c == TAB) {
                return NO;      /* tab is regular */
        }
        else if (c>=127) {
                return YES;     /* del or high bit on */
        }
        else if (c < 32) {
                return YES;     /* control char */
        }
        else {
                return NO;      /* normal */
        }
}

/*
 * handle the default actions of all special keys.
 * return YES if c is one of the keys.
 */

special(c) char c;
{
int k;
        if (c == JOIN1) {
                edjoin();
                pmtline();
                return YES;
        }
        if (c == SPLT1) {
                edsplit();
                pmtline();
                return YES;
        }
        if (c == ABT1) {
                edabt();
                pmtcol();
                return YES;
        }
        else if (c == DEL1) {
                eddel();
                pmtcol();
                return YES;
        }
        else if (c == ZAP1) {
                edzap();
                pmtline();
                return YES;
        }
        else if (c == UP2) {
                /* move up */
                edup();
                pmtline();
                return YES;
        }
        else if (c == UP1) {
                /* insert up */
                ednewup();
                pmtline();
                return YES;
        }
        else if (c == DOWN2) {
                /* move down */
                eddn();
                pmtline();
                return YES;
        }
        else if (c == DOWN1) {
                /* insert down */
                ednewdn();
                pmtline();
                return YES;
        }
        else if (c == LEFT1) {
                edleft();
                pmtcol();
                return YES;
        }
        else if (c == RIGHT1) {
                edright();
                pmtcol();
                return YES;
        }
        else {
                return NO;
        }
}

/*
 * command() dispatches command routines while
 * in command mode.
 */

command()
{
int v;
char c;
char args [SCRNW1];
char *argp;
int topline;
int ypos;
int oldline;
int k;
        /* command mode commands may move the current line.
         * command mode must save the current line on entry
         * and restore it on exit.
         */
        edrepl();
        /* remember how the screen was drawn on entry */
        oldline=bufln();
        ypos=outgety();
        topline=oldline-ypos+1;

        /* Set disk error recovery point. */
        setjmp(D_ERROR);

        while(1) {
                syswait();
                outxy(0,SCRNL1);
                fmtcrlf();
                pmtmode("command:");
                getcmnd(args,0);
                fmtcrlf();
                pmtline();
                c=args [0];
                if ( (c == EDIT1) | (c==INS1) ) {
                        /* redraw screen */
                        if (oldline == bufln()) {
                                /* get current line */
                                edgetln();
                                /* redraw old screen */
                                bufout(topline,1,SCRNL1);
                                outxy(0,ypos);
                                syswait();
                        }
                        else {
                                /* update line and screen */
                                edgo(bufln(),0);
                                syswait();
                        }
                        if (c == EDIT1) {
                                return (EDITMODE);
                        }
                        else {
                                return (INSMODE);
                        }
                }
                else if (tolower(args [0]) == 'g'){
                        argp=skipbl(args+1);
                        if (argp [0] == EOS) {
                                edgo(oldline,0);
                                return EDITMODE;
                        }
                        else if (number(argp,&v) == YES) {
                                edgo(v,0);
                                return EDITMODE;
                        }
                        else {
                                message("bad line number");
                        }
                }
```

(Continued on page 70)

RED - Listing

(Listing continued, text begins on page 62)

```
                else if (lookup(args,"append")) {
                        append(args);
                }
                else if (lookup(args,"change")) {
                        change(args);
                }
                else if (lookup(args,"clear")) {
                        clear();
                }
                else if (lookup(args, "copy")) {
                        copy(args);
                }
                else if (lookup(args,"delete")) {
                        delete(args);
                }
                else if (lookup(args,"dos")) {
                        if (chkbuf() == YES) {
                                /* clean up any temp files. */
                                bufend();
                                return (EXITMODE);
                        }
                }
                else if (lookup(args,"find")) {
                        if ((k = find()) >= 0) {
                                edgo(bufln(),k);
                                return EDITMODE;
                        }
                        else {
                                /* get current line */
                                bufgo(oldline);
                                edgetln();

                                /* stay in command mode */
                                message("pattern not found");
                        }
                }
                else if (lookup(args, "help")) {
                        help();
                }
                else if (lookup(args,"list")) {
                        list(args);
                }
                else if (lookup(args,"load")) {
                        load(args);
                }
                else if (lookup(args,"move")) {
                        move(args);
                }
                else if (lookup(args,"name")) {
                        name(args);
                }
                else if (lookup(args,"resave")) {
                        resave();
                }
                else if (lookup(args,"save")) {
                        save();
                }
                else if (lookup(args,"search")) {
                        search(args);
                }
                else if (lookup(args,"tabs")) {
                        tabs(args);
                }
                else if (lookup(args,"")) {
                        ;
                }
                else {
                        message("command not found");
                }
        }
}

/* return YES if line starts with command */

lookup(line,command) char *line, *command;
{
        while(*command) {
                if (tolower(*line++) != *command++) {
                        return NO;
                }
        }
        if((*line == EOS) | (*line == ' ') | (*line == TAB)) {
                return YES;
        }
        else {
                return NO;
        }
}

/* get next command into argument buffer */

getcmnd(args,offset) char *args; int offset;
{
int j,k;
char c;
        outxy(offset,outgety());
        outdeol();
        k=0;

        while ((c=syscin()) != CR) {
                if ( (c == EDIT1) | (c == INS1) ) {
                        args [0]=c;
                        return;
                }
                if ( (c == DEL1) | (c == LEFT1) ) {
                        if (k>0) {
                                outxy(offset,outgety());
                                outdeol();
                                k--;
                                j=0;
                                while (j < k) {
                                        outchar(args [j++]);
                                }
                        }
                }
                else if (c == ABT1) {
                        outxy(offset,outgety());
                        outdeol();
                        k=0;
                }
                else if ((c != TAB) & ((c < 32)|(c == 127))) {
                        /* do nothing */
                        continue;
                }
                else {
                        if (k+offset < SCRNW1) {
                                args [k++]=c;
                                outchar(c);
                        }
                }
        }
        args [k]=EOS;
}
```

```
/*
        RED command mode commands -- small-C version

        Source:  red3.sc
        Version: January 23, 1983;  February 26, 1983

        Copyright (C) 1983 by Edward K. Ream
*/
/* Define data global to these routines. */

char filename [SYSFNMAX];


/*
        Append command.
        Load a file into main buffer at current location.
        This command does NOT change the current file name.
*/

append(args)
char *args;
{
        char buffer [MAXLEN];   /* disk line buffer */
        int file;
        int n;
        int topline;
        char locfn [SYSFNMAX];  /* local file name */

        /* Get file name which follows command. */
        if (name1(args,locfn) == ERR) {
                return;
        }
        if (locfn [0] == EOS) {
                message("no file argument");
                return;
        }

        /* Open the new file. */
        if ((file = sysopen(locfn, 0)) == ERR) {
                message("file not found");
                return;
        }

        /* Read the file into the buffer. */
        while ((n=sysrdln(file,buffer,MAXLEN)) >= 0) {
                if (n > MAXLEN) {
                        message("line truncated");
                        bufdump();
                        exit();
                        n=MAXLEN;
                }
                bufins(buffer,n);
                bufdn();
        }

        /* Close the file. */
        sysclose(file);

        /*
                Redraw the screen so topline will be at top
                of the screen after command() does a CR/LF.
        */
        topline=max(1,bufln()-SCRNL2);
        bufout(topline,2,SCRNL2);
        bufgo(topline);
}


/* Global change command. */

change(args)
char *args;
{
        char oldline [MAXLEN1]; /* reserve space for EOS */
        char newline [MAXLEN1];
        char oldpat [MAXLEN1];
        char newpat [MAXLEN1];
        int from, to, col, n, k;

        /* Check the arguments. */
        if (get2args(args,&from,&to) == ERR) {
                return;
        }

        /* get search and change masks into oldpat, newpat */
        fmtsout("search mask ?  ",0);
        getcmnd(oldpat,15);
        fmtcrlf();
```

```
        if (oldpat [0] == EOS) {
                return;
        }
        pmtline();
        fmtsout("change mask ?  ",0);
        getcmnd(newpat,15);
        fmtcrlf();

        /* make substitution for lines between from, to */
        while (from <= to) {
                if (chkkey() == YES) {
                        break;
                }
                bufgo(from++);
                if (bufatbot() == YES) {
                        break;
                }
                n=bufgetln(oldline,MAXLEN);
                n=min(n,MAXLEN);
                oldline [n]=EOS;

                /* '^' anchors search */
                if (oldpat [0] == '^') {
                        if (amatch(oldline,oldpat+1,0) == YES) {
                                k=replace(oldline,newline,
                                        oldpat+1,newpat,0);
                                if (k == ERR) {
                                        return;
                                }
                                fmtcrlf();
                                putdec(bufln(),5);
                                fmtsout(newline,5);
                                outdeol();
                                bufrepl(newline,k);
                        }
                        continue;
                }

                /* search oldline for oldpat */
                col=0;
                while (col < n) {
                        if (amatch(oldline,oldpat,col++) == YES){
                                k=replace(oldline,newline,
                                        oldpat,newpat,col-1);
                                if (k == ERR) {
                                        return;
                                }
                                fmtcrlf();
                                putdec(bufln(),5);
                                fmtsout(newline,5);
                                outdeol();
                                bufrepl(newline,k);
                                break;


                        }
                }
        }
        fmtcrlf();
}

/* clear main buffer and file name */

clear()
{
        /* make sure it is ok to clear buffer */
        if (chkbuf() == YES) {
                filename [0]=0;
                pmtfile("");
                outclr();
                outxy(0,SCRNL1);
                bufnew();
                message("buffer cleared");


        }
}

/* Block copy command. */

copy(args)
char * args;
{
        int i, k;
        int last;
        int fstart, fend, tstart;
        char buffer [MAXLEN1];

        /* Get exactly three args. */
        if (get3args(args, &fstart, &fend, &tstart) == ERR) {
                return;


        }

        /*
                The 'to' and 'from' blocks must not overlap.
                Fstart must be > 0, tstart must be >= 0.
        */
        if ( (fend < fstart)  |
             (fstart <= 0)    |
             (tstart < 0)     |
             ( (tstart >= fstart) & (tstart < fend) )
           ) {
                message("oops.  check the copy parameters");
                return;
        }

        /* Make sure the last line exists. */
        last = max(tstart, fstart);

        bufgo(last);

        if (bufln() != last) {
                message("Last line doesn't exist.");
                return;
        }

        /*
                Move the 'from block' to the 'to block'.
                Move one line at a time.
        */
        i = 0;
        while (i <= fend - fstart) {

                /* Go to next line of 'from block'. */
                if (fstart < tstart) {
                        bufgo(fstart + i);
                }
                else {
                        bufgo(fstart + i + i);
                }

                if (bufatbot()) {
                        /* end of 'from block' */
                        break;
                }

                /* Get line of 'from block' into buffer. */
                k = bufgetln(buffer, MAXLEN);

                /* Go to next line of 'to block'. */
                bufgo(tstart + i + 1);

                /* Insert next line into 'to block'. */
                bufins(buffer, k);

                /* Bump the count. */
                i++;
        }
}


/* multiple line delete command */

delete(args)
char *args;
{
        int from, to;

        /* Check the request. */
        if (get2args(args,&from,&to) == ERR) {
                return;
        }
        if (from > to) {
                return;
        }

        /* go to first line to be deleted */
        bufgo(from);

        /* delete all lines between from and to */
        bufdeln(to-from+1);

        /* redraw the screen */
        bufout(bufln(),1,SCRNL1);
}

edithelp()
{

message(
"Here is a list of the commands that you can use in edit mode."
); message(
"Control characters (preceeded by ^) may also be used in insert mode."
); message(
"Type help when in command mode for a list of command mode commands."
); message(
""
); message(
"b          beginning of line    ^d      move cursor up; enter edit mode"
); message(
"c or ESC   enter command mode   ^e      enter edit mode"
); message(
"d          scroll down          ^h      delete character"
); message(
"e          end of line          ^l      move cursor left"
); message(
"g <n>      go to line <n>       ^p      join two lines"
); message(
"h          print this message   ^r      move cursor right"
); message(
"i or ^n    enter insert mode    ^s      split line at cursor"
); message(
"k <let>    delete to <let>      ^u      move cursor up; enter edit mode"
); message(
"s <let>    set cursor to <let>  ^x      undo changes to the line"
); message(
"u          scroll up            ^z      delete line"
); message(
"carriage return:   insert line below current line;  enter insert mode"
); message(
"line feed:         insert line above current line;  enter insert mode"
); message(
""
); message(
"Type any character to continue editing... "
);
        pmtedit();
        syscin();
}
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 62)

```
/* search all lines below the current line for a pattern
 * return -1 if pattern not found.
 *                    column number of start of pattern.
 *
                                                    ln() + 1, HUGE, YES);


help()
{

message(
"Here is a list of commands you can use in command mode."
); message(
"Type h when in edit mode for more help."
); message(
""
); message(
"append <filename>     append a file after the current line"
); message(
"change <line range>   change all lines in <line range>"
); message(
"clear                 reset the editor"
); message(
"copy <n1> <n2> <n3>   copy lines <n1> through <n2> after <n3>"
); message(
"delete <line range>   delete all lines in <line range>"
); message(
"dos                   exit from the editor"
); message(
"find                  search for a pattern;  Enter edit mode if found"
); message(
"g <n>                 enter edit mode at line <n>"
); message(
"g or ^e               enter edit mode at the current line"
); message(
"help                  print this message"
); message(
"list                  list lines to the printer"
); message(
"load <filename>       replace the buffer with <filename>"
); message(
"move <n1> <n2> <n3>   move lines <n1> through <n2> after <n3>"
); message(
"name <filename>       set the filename for the save and resave"
); message(
"resave                save the buffer to the already existing file"
); message(
"save                  save the buffer to a new file"
); message(
"search                list all lines which contain a pattern"
); message(
"tabs <n>              set tabs to every <n> columns"
);

}


/* list lines to list device */

list(args)
char *args;
{
        char linebuf [MAXLEN1];
        int n;
        int from, to, line, oldline;

        /* save the buffer's current line */
        oldline=bufln();

        /* get starting, ending lines to print */
        if (get2args(args,&from,&to) == ERR) {
                return;
        }

        /* print lines one at a time to list device */
        line=from;
        while (line <= to) {

                /* make sure prompt goes to console */
                fmtassn(NO);

                /* check for interrupt */
                if (chkkey() == YES) {

                        break;
                }

                /* print line to list device */
                fmtassn(YES);

                bufgo(line++);
                if (bufatbot()) {
                        break;
                }
                n=bufgetln(linebuf,MAXLEN1);
                n=min(n,MAXLEN);
                linebuf [n]=CR;
                fmtsout(linebuf,0);
                fmtcrlf();
        }

        /* redirect output to console */
        fmtassn(NO);

        /* restore cursor */
        bufgo(oldline);
}


/* Load file into buffer. */

load (args)
char *args;
{
        char buffer [MAXLEN];   /* disk line buffer */
        char locfn  [SYSFNMAX];  /* file name */
        int n;
        int topline;

        /* Get filename following command. */
        if (name1(args,locfn) == ERR) {
                return;
        }

        if (locfn [0] == EOS) {
                message("no file argument");
                return;
        }

        /* Give user a chance to save the buffer. */
        if (chkbuf() == NO) {
                return;
        }

        /* Open the new file. */
        if (sysexists(locfn) == NO) {
                message("file not found");
                return;
        }

        /* Update file name. */
        syscopfn(locfn, filename);
        pmtfile(filename);

        /* Clear the buffer. */
        bufnew();

        /* Read the whole file into the buffer. */
        buf_r_file(filename);
        /* indicate that the buffer is fresh */
        bufsaved();

        /* set current line to line 1 */
        bufgo(1);

        /*
                Redraw the screen so that topline will be
                on line 1 after command() does a CR/LF.
        */
        topline=max(1,bufln()-SCRNL2);
        bufout(topline,2,SCRNL2);
        bufgo(topline);
}


/* Block move command. */

move(args)
char * args;
{
        int c, i, k;
        int last;
        int fstart, fend, tstart;
        char buffer [MAXLEN1];

        /* Get exactly three args. */
        if (get3args(args, &fstart, &fend, &tstart) == ERR) {
                return;
        }

        /*
                The 'to' and 'from' blocks must not overlap.
                Fstart must be > 0, tstart must be >= 0.
        */
        if ( (fend < fstart)  |
             (fstart <= 0)    |
             (tstart < 0)     |
             ( (tstart >= fstart) & (tstart <= fend) )
           ) {
                message("oops. check the move parameters");
                return;
        }

        /* Make sure the last line exists. */
        if (tstart < fstart) {
                last = fstart;
        }
        else {
                last = tstart;
        }

        bufgo(last);
        if (bufln() != last) {
                return;
        }

        /*
                Move the 'from block' to the 'to block'.
                Move one line at a time.
        */
        i = c = 0;
        while (c++ <= fend - fstart) {

                /* Go to next line of 'from block'. */
                bufgo(fstart + i);

                if (bufatbot()) {
                        /* end of 'from block' */
                        break;
                }
```

```
                /* Get line of 'from block' into buffer. */
                k = bufgetln(buffer, MAXLEN);

                /* Delete this line. */
                bufdeln(1);
                /* Go to next line of 'to block'. */
                if (tstart < fstart) {

                        /* Delete leaves 'to block' numbers. */
                        bufgo(tstart + i + 1);
                }
                else {

                        /* Delete decreased numbers by one. */
                        bufgo(tstart + i);
                }

                /* Insert next line into 'to block'. */
                bufins(buffer, k);

                /* Adjust line numbers if needed. */
                if (tstart < fstart) {

                        /* Line numbers increase. */
                        i++;
                }
        }
}


/* change current file name */

name(args)
char *args;
{
        name1(args,filename);
        pmtfile(filename);
}


/* check syntax of args.
 * copy to filename.
 * return OK if the name is valid.
 */

name1(args,filename)
char *args, *filename;
{
        /* skip command */
        args=skiparg(args);
        args=skipbl(args);

        /* check file name syntax */
        if (syschkfn(args) == ERR) {
                return ERR;
        }

        /* copy filename */
        syscopfn(args,filename);

        return OK;
}


/* Save the buffer in an already existing file. */

resave()
{
        int n, oldline;

        /* Save line number. */
        oldline = bufln();

        /* Make sure file has a name. */
        if (filename [0] == EOS) {
                message("file not named");
                return;
        }

        /* The file must exist for resave. */
        if (sysexists(filename) == NO) {
                message("file not found");
                return;
        }

        /* Write out the whole buffer. */
        buf_w_file(filename);

        /* Indicate that the buffer has been saved. */
        bufsaved();

        /* Restore line number. */
        bufgo(oldline);
}
/* Save the buffer in a new file. */

save()
{
        int file, n, oldline;

        /* Save current line number. */
        oldline = bufln();

        /* Make sure the file is named. */
        if (filename [0] == EOS) {
                message("file not named");
                return;
        }

        /* File must NOT exist for save. */
        if (sysexists(filename) == YES) {
                message("file exists");
                return;
        }

        /* Write out the whole buffer. */
        buf_w_file(filename);

        /* Indicate buffer saved. */
        bufsaved();

        /* Restore line number. */
        bufgo(oldline);
}
/* global search command */

search(args)
char *args;
{
int from, to;

        /* Check the request. */
        if (get2args(args,&from,&to) == ERR) {
                return;
        }
        search1(from, to, NO);
}

/* search lines for a pattern.
 * if flag  ==  YES: stop at the first match.
 *                   return -1 if no match.
 *                   otherwise return column number of match.
 * if flag  ==  NO:  print all matches found.
 */

search1(from, to, flag)
int from, to, flag;
{
        char pat   [MAXLEN1];   /* reserve space for EOS */
        char line  [MAXLEN1];
        int col, n;

        /* get search mask into pat */
        fmtsout("search mask ?  ",0);
        getcmnd(pat,15);
        fmtcrlf();

        if (pat [0] == EOS) {
                return -1;      /* bug fix */
        }

        /* search all lines between from and to for pat */
        while (from <= to) {
                if (chkkey() == YES) {
                        break;
                }
                bufgo(from++);
                if (bufatbot() == YES) {
                        break;
                }
                n=bufgetln(line,MAXLEN);
                n=min(n,MAXLEN);
                line [n]=EOS;

                /* ^ anchors search */
                if (pat [0] == '^') {
                        if (amatch(line,pat+1,0) == YES) {
                                if (flag == NO) {
                                        fmtcrlf();
                                        putdec(bufln(),5);
                                        fmtsout(line,5);
                                        outdeol();
                                }
                                else {
                                        return 0;
                                }
                        }

                        continue;
                }

                /* search whole line for match */
                col=0;
                while (col < n) {
                        if (amatch(line,pat,col++) == YES) {
                                if (flag == NO) {
                                        fmtcrlf();
                                        putdec(bufln(),5);
                                        fmtsout(line,5);
                                        outdeol();
                                        break;
                                }
                                else {
                                        return col-1;
                                }
                        }
                }
        }

        /* all searching is finished */
        if (flag == YES) {
                return -1;
        }
        else {
                fmtcrlf();
        }
}

/* set tab stops for fmt routines */

tabs(args)
char *args;
{
        int n, junk;

        if (get2args(args,&n,&junk) == ERR) {
                return;
        }
        fmtset(n);
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 62)

```
}


/* return YES if buffer may be drastically changed */

chkbuf()
{
        if (bufchng() == NO) {

                /* buffer not changed. no problem */
                return YES;
        }

        fmtsout("buffer not saved. proceed ?  ",0);
        pmtline();

        if (tolower(syscout(syscin())) != 'y') {
                fmtcrlf();
                message("cancelled");
                return NO;
        }
        else {
                fmtcrlf();
                return YES;
        }

}
/* print message from a command */

message(s)
char *s;
{
        fmtsout(s,0);
        fmtcrlf();
}
/* get two arguments the argument line args.
 * no arguments imply 1 HUGE.
 * one argument implies both args the same.
 */
get2args(args,val1,val2)
char *args;
int *val1, *val2;
{
        /* skip over the command */
        args=skiparg(args);
        args=skipbl(args);

        if (*args == EOS) {
                *val1=1;
                *val2=HUGE;
                return OK;
        }
        /* check first argument */
        if (number(args,val1) == NO) {
                message("bad argument");
                return ERR;
        }

        /* skip over first argument */
        args=skiparg(args);
        args=skipbl(args);

        /* 1 arg: arg 2 is HUGE */
        if (*args == EOS) {
                *val2=HUGE;
                return OK;
        }

        /* check second argument */
        if (number(args,val2) == NO) {
                message("bad argument");
                return ERR;
        }
        else {
                return OK;
        }
}

/* Get exactly three arguments. */

get3args(args, val1, val2, val3)
char *args;
int *val1, *val2, *val3;
{
        /* Skip the command. */
        args = skiparg (args);
        args = skipbl (args);

        /* Check first arg. */
        if (*args == EOS) {
                message("missing arguments");
                return ERR;
        }

        if (number (args, val1) == NO) {
                message("bad argument");
                return ERR;
        }

        /* Skip over first argument. */
        args = skiparg(args);
        args = skipbl(args);

        /* Check second argument. */
        if (*args == EOS) {
                message("missing arguments");
                return ERR;
        }

        if (number(args, val2) == NO) {
                message("bad argument");
                return ERR;
        }

        /* Skip over third argument. */
        args = skiparg(args);
        args = skipbl(args);

        /* Check third argument. */
        if (*args == EOS) {
                message("missing arguments");
                return ERR;
        }

        if (number (args, val3) == NO) {
                message("bad argument");
                return ERR;
        }
        else {
                return OK;
        }
}


/* skip over all except EOS, and blanks */

skiparg(args) char *args;
{
        while ( (*args != EOS) & (*args!=' ') ) {
                args++;
        }
        return args;
}


/* skip over all blanks */

skipbl(args) char *args;
{
        while (*args == ' ') {
                args++;

        }
        return args;
}


/* return YES if the user has pressed any key.
 * blanks cause a transparent pause.
 */

chkkey()
{
        int c;

        c=syscstat();
        if (c == -1) {          /* bug fix */

                /* no character at keyboard */
                return NO;
        }
        else if (c == ' ') {

                /* pause.  another blank ends pause */
                pmtline();
                if (syscin() == ' ') {
                        return NO;
                }
        }

        /* we got a nonblank character */
        return YES;
}


/* anchored search for pattern in text line at column col.
 * return YES if the pattern starts at col.
 */

amatch(line,pat,col)
char *line, *pat;
int col;
{
        int k;

        k=0;
        while (pat [k] != EOS) {
                if (pat [k] == line[col]) {
                        k++;
                        col++;
                }
                else if ((pat [k] == '?')&(line[col] != EOS)) {

                        /* question mark matches any char */
                        k++;
                        col++;
                }
                else {
                        return NO;
                }
        }
```

```
        /* the entire pattern matches */
        return YES;
}


/* replace oldpat in oldline by newpat starting at col.


 * put result in newline.
 * return number of characters in newline.
 */

replace(oldline,newline,oldpat,newpat,col)
char *oldline, *newline, *oldpat, *newpat;
int col;
{
        int k;
        char *tail, *pat;

        /* copy oldline preceding col to newline */
        k=0;
        while (k < col) {
                newline [k++]=*oldline++;
        }

        /* remember where end of oldpat in oldline is */
        tail=oldline;
        pat=oldpat;
        while (*pat++ != EOS) {
                tail++;
        }

        /* copy newpat to newline.
         * use oldline and oldpat to resolve question marks
         * in newpat.
         */
        while (*newpat != EOS) {
                if (k > MAXLEN-1) {
                        message("new line too long");
                        return ERR;
                }
                if (*newpat != '?') {
                        /* copy newpat to newline */
                        newline [k++]=*newpat++;
                        continue;
                }

                /* scan for '?' in oldpat */
                while (*oldpat != '?') {
                        if (*oldpat == EOS) {
                                message(
                                "too many ?'s in change mask"
                                );
                                return ERR;
                        }
                        oldpat++;
                        oldline++;
                }

                /* copy char from oldline to newline */
                newline [k++]=*oldline++;
                oldpat++;
                newpat++;
        }

        /* copy oldline after oldpat to newline */
        while (*tail != EOS) {
                if (k >= MAXLEN-1) {
                        message("new line too long");
                        return ERR;
                }
                newline [k++]=*tail++;
        }
        newline [k]=EOS;

        return k;
}
```

```
/*
        RED window module -- small-C version

        Source:  red4.sc
        Version: January 23, 1983

        Copyright (C) 1983 by Edward K. Ream
*/
/* data global to this module */

char    editbuf[MAXLEN];        /* the edit buffer      */
int     editp;                  /* cursor: buffer index */
int     editpmax;               /* length of buffer     */
int     edcflag;                /* buffer change flag   */

/* abort any changes made to current line */
edabt()
{
        /* get unchanged line and reset cursor */
        edgetln();
        edredraw();
        edbegin();
        edcflag = NO;
}
/* put cursor at beginning of current line */

edbegin()
{
        editp = 0;
        outxy(0,outgety());
}

/* change editbuf[editp] to c
 * don't make change if line would become to long
 */
edchng(c) char c;
{
        char oldc;
        int k;
        /* if at right margin then insert char */
        if (editp >= editpmax) {
                edins(c);
                return;
        }
        /* change char and print length of line */
        oldc = editbuf[editp];
        editbuf[editp] = c;
        fmtadj(editbuf,editp,editpmax);
        k = fmtlen(editbuf,editp+1);
        if (k > SCRNW1) {

                /* line would become too long */
                /* undo the change */
                editbuf[editp] = oldc;
                fmtadj(editbuf,editp,editpmax);
        }
        else {

                /* set change flag, redraw line */
                edcflag = YES;
                editp++;
                edadj();
                edredraw();
        }
}

/* delete the char to left of cursor if it exists */

eddel()
{
        int k;

        /* just move left one column if past end of line */
        if (edxpos() < outgetx()) {
                outxy(outgetx()-1, outgety());
                return;
        }

        /* do nothing if cursor is at left margin */
        if (editp == 0) {
                return;
        }
        edcflag = YES;
        /* compress buffer (delete char) */
        k = editp;
        while (k < editpmax) {
                editbuf[k-1] = editbuf[k];
                k++;
        }

        /* update pointers, redraw line */
        editp--;
        editpmax--;
        edredraw();
}

/* edit the next line.  do not go to end of buffer */
eddn()
{
        int oldx;
        /* save visual position of cursor */
        oldx = outgetx();

        /* replace current edit line */
        edrepl();

        /* do not go past last non-null line */
        if (bufnrbot()) {
                return;
        }

        /* move down one line in buffer */
        bufdn();
        edgetln();
        /* put cursor as close as possible on this
         * new line to where it was on the old line.
         */
        editp = edscan(oldx);

        /* update screen */
        if (edatbot()) {
                edsup(bufln()-SCRNL2);
                outxy(oldx, SCRNL1);
        }
        else {
                outxy(oldx, outgety()+1);
        }
        return;
}

/* put cursor at the end of the current line */

edend()
{
        editp = editpmax;
        edadj();
        outxy(edxpos(),outgety());
}


/* start editing line n
 * redraw the screen with cursor at position p
 */
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 62)

```
edgo(n, p) int n, p;
{
        /* replace current line */
        edrepl();

        /* go to new line */
        bufgo(n);

        /* prevent going past end of buffer */
        if (bufatbot()) {
                bufup();
        }

        /* redraw the screen */
        bufout(bufln(),1,SCRNL1);
        edgetln();
        editp = min(p, editpmax);
        outxy(edxpos(), 1);
        return;
}


/* insert c into the buffer if possible */

edins(c)
char c;
{
        int k;

        /* do nothing if edit buffer is full */
        if (editpmax >= MAXLEN) {
                return;
        }

        /* fill out line if we are past its end */
        if ((editp == editpmax) & (edxpos() < outgetx())) {
                k = outgetx() - edxpos();
                editpmax = editpmax + k;
                while (k-- > 0) {

                        editbuf [editp++] = ' ';
                }
                editp = editpmax;
        }

        /* make room for inserted character */
        k = editpmax;
        while (k > editp) {
                editbuf[k] = editbuf[k-1];
                k--;
        }

        /* insert character, update pointers */
        editbuf[editp] = c;
        editp++;
        editpmax++;

        /* recalculate print length of line  */
        fmtadj(editbuf,editp-1,editpmax);
        k = fmtlen(editbuf,editp);
        if ( (k > SCRNW1) & (editp == editpmax) ) {
                /* auto-split the line (line wrap) */

                /* scan for the start of the current word */
                k = editp - 1;
                while ( (k >= 0) &
                        (editbuf[k] != ' ') &
                        (editbuf[k] != TAB)
                      ) {
                        k--;
                }

                /* never split a word */
                if (k < 0) {
                        eddel();
                        return;
                }

                /* split the line at the current word */
                editp = k + 1;
                edsplit();
                edend();
        }
        else if (k > SCRNW1) {

                /* line would become too long */
                /* delete what we just inserted */
                eddel();
        }
        else {
                /* set change flag, redraw line */
                edcflag = YES;
                edredraw();
        }
}


/* join (concatenate) the current line with the one above it */

edjoin()
{
        int k, k1, k2;

        /* do nothing if at top of file */
        if (bufattop()) {
                return;

        }

        /* replace lower line temporarily */
        edrepl();

        /* get upper line into buffer */
        bufup();
        k1 = bufgetln(editbuf, MAXLEN);

        /* append lower line to buffer */
        bufdn();
        k2 = bufgetln(editbuf+k1, MAXLEN-k1);

        /* abort if the screen isn't wide enough */
        if (k1 + k2 > SCRNW1) {

                /* bug fix */
                bufgetln(editbuf,MAXLEN);
                return;
        }

        /* replace upper line */
        bufup();
        editpmax = k1 + k2;
        editp = k1 + editp;
        edadj();
        edcflag = YES;
        edrepl();

        /* delete the lower line */
        bufdn();
        bufdel();
        bufup();

        /* update the screen */
        if (edattop()) {
                edredraw();
        }
        else {
                k = outgety() - 1;
                bufout(bufln(),k,SCRNL-k);
                outxy(0,k);
                edredraw();
        }
}


/* delete chars until end of line or c found */

edkill(c) char c;
{

        int k,p;

        /* do nothing if at right margin */
        if (editp == editpmax) {
                return;
        }
        edcflag = YES;

        /* count number of deleted chars */
        k = 1;
        while ((editp+k) < editpmax) {
                if (editbuf[editp+k] == c) {
                        break;
                }
                else {
                        k++;
                }
        }

        /* compress buffer (delete chars) */
        p = editp+k;
        while (p < editpmax) {
                editbuf[p-k] = editbuf[p];
                p++;
        }

        /* update buffer size, redraw line */
        editpmax = editpmax-k;
        edredraw();
}

/* move cursor left one column.
 * never move the cursor off the current line.
 */

edleft()
{
int k;

        /* if past right margin, move left one column */
        if (edxpos() < outgetx()) {
                outxy(max(0, outgetx()-1), outgety());
        }

        /* inside the line.  move left one character */
        else if (editp != 0) {
                editp--;
                outxy(edxpos(),outgety());
        }
}


/* insert a new blank line below the current line */

ednewdn()
{
        int k;

        /* make sure there is a current line and
         * put the current line back into the buffer.
         */
```

```
        if (bufatbot()) {
                bufins(editbuf,editpmax);
        }
        edrepl();

        /* move past current line */
        bufdn();

        /* insert place holder:  zero length line */
        bufins(editbuf,0);

        /* start editing the zero length line */
        edgetln();

        /* update the screen */
        if (edatbot()) {

                /* note: bufln()  >= SCRNL */
                edsup(bufln()-SCRNL2);
                outxy(edxpos(),SCRNL1);



        }
        else {
                k = outgety();
                bufout(bufln(),k+1,SCRNL1-k);
                outxy(edxpos(),k+1);
        }
}


/* insert a new blank line above the current line */

ednewup()
{
int k;
        /* put current line back in buffer */
        edrepl();

        /* insert zero length line at current line */
        bufins(editbuf,0);

        /* start editing the zero length line */
        edgetln();

        /* update the screen */
        if (edattop()) {
                edsdn(bufln());
                outxy(edxpos(),1);
        }
        else {
                k = outgety();
                bufout(bufln(),k,SCRNL-k);
                outxy(edxpos(),k);
        }
}


/* move cursor right one character.
 * never move the cursor off the current line.
 */

edright()

{
        /* if we are outside the line move right one column */
        if (edxpos() < outgetx()) {
                outxy (min(SCRNW1, outgetx()+1), outgety());
        }

        /* if we are inside a tab move to the end of it */
        else if (edxpos() > outgetx()) {
                outxy (edxpos(), outgety());
        }

        /* move right one character if inside line */
        else if (editp < editpmax) {
                editp++;
                edadj();
                outxy(edxpos(),outgety());
        }

        /* else move past end of line */
        else {
                outxy (min(SCRNW1, outgetx()+1), outgety());
        }
}

/* split the current line into two parts.
 * scroll the first half of the old line up.
 */

edsplit()
{
        int p, q;
        int k;

        /* indicate that edit buffer has been saved */
        edcflag = NO;

        /* replace current line by the first half of line */
        if (bufatbot()) {
                bufins(editbuf, editp);
        }
        else {
                bufrepl(editbuf, editp);
        }

        /* redraw the first half of the line */
        p = editpmax;
        q = editp;
        editpmax = editp;
        editp = 0;
        edredraw();

        /* move the second half of the line down */
        editp = 0;
        while (q < p) {
                editbuf [editp++] = editbuf [q++];
        }
        editpmax = editp;
        editp = 0;

        /* insert second half of the line below the first */
        bufdn();
        bufins(editbuf, editpmax);

        /* scroll the screen up and draw the second half */
        if (edatbot()) {
                edsup(bufln()-SCRNL2);
                outxy(1,SCRNL1);
                edredraw();
        }
        else {
                k = outgety();
                bufout(bufln(), k+1, SCRNL1-k);
                outxy(1, k+1);
                edredraw();
        }
}


/* move cursor right until end of line or
 * character c found.
 */

edsrch(c) char c;
{
        /* do nothing if at right margin */
        if (editp == editpmax) {
                return;
        }

        /* scan for search character */

        editp++;
        while (editp < editpmax) {
                if (editbuf[editp] == c) {
                        break;
                }
                else {
                        editp++;
                }
        }

        /* reset cursor */
        edadj();
        outxy(edxpos(),outgety());
}


/* move cursor up one line if possible */

edup()
{
        int oldx;

        /* save visual position of cursor */
        oldx = outgetx();

        /* put current line back in buffer */
        edrepl();

        /* done if at top of buffer */
        if (bufattop()) {
                return;
        }

        /* start editing the previous line */
        bufup();
        edgetln();

        /* put cursor on this new line as close as
         * possible to where it was on the old line.
         */
        editp = edscan(oldx);

        /* update screen */
        if (edattop()) {
                edsdn(bufln());
                outxy(oldx, 1);
        }
        else {
                outxy(oldx, outgety()-1);
        }
        return;
}

/* delete the current line */

edzap()
{
        int k;

        /* delete the line in the buffer */
        bufdel();

        /* move up one line if now at bottom */
        if (bufatbot()) {
```

(Continued on page 82)

RED - Listing

(Listing continued, text begins on page 62)

```
                bufup();
                edgetln();
                /* update screen */
                if (edattop()) {
                        edredraw();
                }
                else {
                        outdelln();
                        outxy(0,outgety()-1);
                }
                return;
        }

        /* start editing new line */
        edgetln();

        /* update screen */
        if (edattop()) {
                edsup(bufln());
                outxy(0,1);
        }
        else {
                k = outgety();
                bufout(bufln(),k,SCRNL-k);
                outxy(0,k);
        }
}

/* ----- utility routines (not used outside this file) ----- */

/* adjust the cursor so it stays on the screen.
 * call this routine whenever the cursor could move right.
 */

edadj()
{
        while (fmtlen(editbuf, editp) > SCRNW1) {
                editp--;
        }
}


/* return true if the current edit line is being
 * displayed on the bottom line of the screen.
 */

edatbot()
{
        return outgety() == SCRNL1;
}


/* return true if the current edit line is being
 * displayed on the bottom line of the screen.
 */

edattop()
{
        return outgety() == 1;
}


/* redraw edit line from index to end of line */
/* reposition cursor */
edredraw()
{
        fmtadj(editbuf,0,editpmax);
        fmtsubs(editbuf,max(0,editp-1),editpmax);
        outxy(edxpos(),outgety());
}


/* return the x position of the cursor on screen */

edxpos()
{
        return fmtlen(editbuf, editp);
}


/* fill edit buffer from current main buffer line.
 * the caller must check to make sure the main
 * buffer is available.
 */

edgetln()
{
        int k;

        /* put cursor on left margin, reset flag */
        editp = 0;
        edcflag = NO;

        /* get edit line from main buffer */
        k = bufgetln(editbuf,MAXLEN);
        if (k > MAXLEN) {
                error("line truncated");
                editpmax = MAXLEN;
        }
        else {
                editpmax = k;
        }
        fmtadj(editbuf,0,editpmax);
}

/* Replace current main buffer line by edit buffer.
 * The edit buffer is NOT changed or cleared.
 */

edrepl()
{
        /* do nothing if nothing has changed */
        if (edcflag == NO) {
                return;
        }

        /* make sure we don't replace the line twice */
        edcflag = NO;

        /* insert instead of replace if at bottom of file */
        if (bufatbot()) {
                bufins(editbuf,editpmax);
        }
        else {
                bufrepl(editbuf,editpmax);
        }
}


/* set editp to the largest index such that
 * buf[editp] will be printed <= xpos
 */

edscan(xpos) int xpos;
{
        editp = 0;
        while (editp < editpmax) {
                if (fmtlen(editbuf,editp) < xpos) {
                        editp++;
                }
                else {
                        break;
                }
        }
        return editp;
}


/* scroll the screen up.  topline will be new top line */

edsup(topline) int topline;
{
        if (outhasup() == YES) {

                /* hardware scroll */
                outsup();

                /* redraw bottom line */
                bufout(topline+SCRNL2,SCRNL1,1);
        }
        else {

                /* redraw whole screen */
                bufout(topline,1,SCRNL1);
        }
}
/* scroll screen down.  topline will be new top line */

edsdn(topline) int topline;
{
        if (outhasdn() == YES) {

                /* hardware scroll */
                outsdn();

                /* redraw top line */
                bufout(topline,1,1);
        }
        else {

                /* redraw whole screen */
                bufout(topline,1,SCRNL1);
        }
}
```

```
/*
        RED output format module -- small-C version

        Source:  red5.sc
        Version: August 21, 1982.

        Copyright (C) 1983 by Edward K. Ream
*/


/* Define variables global to this module. */

/* define maximal length of a tab character */

int fmttab;


/* define the current device and device width */

int fmtdev;             /* device -- YES/NO = LIST/CONSOLE */
int fmtwidth;           /* devide width.  LISTW/SCRNW1 */


/*
     fmtcol[i] is the first column at which
     buf[i] will be printed.
     fmtsub() and fmtlen() assume fmtcol[] is valid on entry.
*/

int fmtcol[MAXLEN1];


/*
        Direct output from this module to either the console or
        the list device.
*/

fmtassn(listflag) int listflag;
```

```
{
        if (listflag==YES) {
                fmtdev=YES;
                fmtwidth=LISTW;
        }
        else {
                fmtdev=NO;
                fmtwidth=SCRNW1;
        }
}


/*
        Adjust fmtcol[] to prepare for calls on
        fmtout() and fmtlen().

        NOTE:  this routine is needed as an efficiency
               measure.  Without fmtadj(), calls on
               fmtlen() become too slow.
*/

fmtadj(buf,minind,maxind) char *buf; int minind,maxind;
{
int k;
        /* line always starts at left margin */
        fmtcol[0]=0;
        /* start scanning at minind */
        k=minind;
        while (k<maxind) {

                /* comment out -----
                if (buf[k]==CR) {
                        break;
                }
                ----- end comment out */

                fmtcol[k+1]=fmtcol[k]+fmtlench(buf[k],fmtcol[k]);
                k++;
        }
}
/* return column at which at which buf[i] will be printed */

fmtlen(buf,i) char *buf; int i;
{
        return(fmtcol[i]);
}


/*
        Print buf[i] ... buf[j-1] on current device so long as
        characters will not be printed in last column.
*/

fmtsubs(buf,i,j) char *buf; int i, j;
{
int k;
        if (fmtcol[i]>=fmtwidth) {
                return;
        }
        outxy(fmtcol[i],outgety());     /* position cursor */
        while (i<j) {

                /* Comment out -----
                if (buf[i]==CR) {
                        break;
                }
                ----- end comment out */

                if (fmtcol[i+1]>fmtwidth) {
                        break;
                }
                fmtoutch(buf[i],fmtcol[i]);
                i++;
        }
        outdeol();      /* clear rest of line */
}
/*
    Print string which ends with CR or EOS to current device.
    Truncate the string if it is too long.
*/

fmtsout(buf,offset) char *buf; int offset;
{
char c;
int col,k;
        col=0;
        while (c=*buf++) {
                if (c==CR) {
                        break;
                }
                k=fmtlench(c,col);
                if ((col+k+offset)>fmtwidth) {
                        break;
                }
                fmtoutch(c,col);
                col=col+k;
        }
}
/* Return length of char c at column col. */

fmtlench(c,col) char c; int col;
{
        if (c==TAB) {
                /* tab every fmttab columns */
                return(fmttab-(col%fmttab));
        }
        else if (c<32) {
                /* control char */
                return(2);
        }
        else {
                return(1);
        }
}

/*
        Output one character to current device.
        Convert tabs to blanks.
*/

fmtoutch(c,col) char c; int col;
{
int k;
        if (c==TAB) {
                k=fmtlench(TAB,col);
                while ((k--)>0) {
                        fmtdevch(' ');
                }
        }
        else if (c<32) {
                fmtdevch('^');
                fmtdevch(c+64);
        }
        else {
                fmtdevch(c);
        }
}


/* Output character to current device. */

fmtdevch(c) char c;
{
        if (fmtdev==YES) {
                syslout(c & 127);
        }
        else {
                outchar(c & 127);
        }
}


/* Output a CR and LF to the current device. */

fmtcrlf()
{
        if (fmtdev==YES) {
                syslout(CR);
                syslout(LF);
        }
        else {
                /* kludge: this should be in out module */
                /* make sure out module knows position */
                outxy(0,SCRNL1);
                syscout(CR);
                syscout(LF);
        }
}


/* Set tabs at every n columns. */

fmtset(n) int n;
{
        fmttab=max(1,n);
}
```

```
/*
RED terminal output module
Source:  red6.sc
This file was created by the configuration program:
January 20, 1983; February 26, 1983

Modified by hand: gotoxy(), outdeol().
*/

/*
Define the current coordinates of the cursor.
*/

int outx, outy;


/*
Return the current coordinates of the cursor.
*/

outgetx()
{
        return(outx);
}

outgety()
{
        return(outy);
}

/*
Output one printable character to the screen.
*/

outchar(c) char c;
{
        syscout(c);
        outx++;
        return(c);
}

/*
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 62)

```
Position cursor to position x,y on screen.
0,0 is the top left corner.
*/
outxy(x,y) int x,y;
{
                syscout(11);
                syscout(64+y);
                syscout(16);
                syscout( ((x/10)<<4) | (x%10) );
        outx=x;
        outy=y;
}

/*
Erase the entire screen.
Make sure the rightmost column is erased.
*/
outclr()
{
        int k;

        k=0;
        while (k<SCRNL) {
                outxy(0,k++);
                outdelln();
        }
        outxy(0,0);
}

/*
Delete the line on which the cursor rests.
Leave the cursor at the left margin.
*/
outdelln()
{
        outxy(0,outy);
        outdeol();
}

/*
Delete to end of line.
Assume the last column is blank.
*/
outdeol()
{
        syscout(27);
        syscout('K');
}

/*
Return yes if terminal has indicated hardware scroll.
*/
outhasup()
{
        return(YES);
}

outhasdn()
{
        return(YES);
}

/*
Scroll the screen up.
Assume the cursor is on the bottom line.
*/
outsup()
{
        /* auto scroll */
        outxy(0,SCRNL1);
        syscout(10);
}

/*
Scroll screen down.
Assume the cursor is on the top line.
*/
outsdn()
{
        int wait;

        /* auto scroll */
        outxy(0,0);
        syscout(27);
        syscout('I');

        /* 19.2 Kbaud kludge */
        wait = 0;
        while (wait++ < 150) {
                ;
        }
}
```

```
/*
        RED prompt line module -- small-C version

        Source:  red7.sc
        Version: August 24, 1982.

        Copyright (C) 1983 by Edward K. Ream
*/
/* Define the prompt line data. */

char pmtln[MAXLEN];     /* mode */
char pmtfn[SYSFNMAX];   /* file name */

/* Initialize the mode and file name. */

pmtclr()
{
        pmtln [0] = 0;
        pmtfn [0] = 0;
}


/*
        Put error message on prompt line.
        Wait for response.
*/

pmtmess(s1,s2)
char *s1, *s2;
{
        int x,y;

        /* save cursor */
        x=outgetx();
        y=outgety();
        outxy(0,0);
        /* make sure line is correct */
        outdelln();
        pmtline1();
        pmtcol1(x);
        /* output error message */
        fmtsout(s1,outgetx());
        fmtsout(s2,outgetx());
        /* wait for input from console */
        syscin();
        /* redraw prompt line */
        pmtline1();
        pmtcol1(x);
        pmtfile1(pmtfn);
        pmtmode1(pmtln);
        /* restore cursor */
        outxy(x,y);
}


/* Write new mode message on prompt line. */

pmtmode(s)
char *s;
{
        int x,y;                /* save cursor on entry */

        /* save cursor */
        x=outgetx();
        y=outgety();

        /* redraw whole line */
        outxy(0,0);
        outdelln();
        pmtline1();
        pmtcol1(x);
        pmtfile1(pmtfn);
        pmtmode1(s);
        /* restore cursor */
        outxy(x,y);
}


/* Update file name on prompt line. */

pmtfile(s)
char *s;
{
        int x, y;

        /* save cursor */
        x=outgetx();
        y=outgety();
        /* update whole line */
        outxy(0,0);
        outdelln();
        pmtline1();
        pmtcol1(x);             /* bug fix -- 1/28/82 */
        pmtfile1(s);
        pmtmode1(pmtln);
        /* restore cursor */
        outxy(x,y);
}


/* Change mode on prompt line to edit: */

pmtedit()
{
        pmtmode("edit:");
}


/* Update line and column numbers on prompt line. */

pmtline()
{
        int x,y;

        /* save cursor */
        x=outgetx();
        y=outgety();
        /* redraw whole line */
        outxy(0,0);
        outdelln();
        pmtline1();
        pmtcol1(x);
        pmtfile1(pmtfn);
        pmtmode1(pmtln);
        /* restore cursor */
        outxy(x,y);
}
```

```
/* Update just the column number on prompt line. */

pmtcol()
{
        int x,y;

        /* save cursor */
        x=outgetx();
        y=outgety();
        /* update column number */
        pmtcol1(x);
        /* update cursor */
        outxy(x,y);
}


/* Update mode.  call getcmnd() to write on prompt line. */

pmtcmnd(mode,buffer)
char *mode, *buffer;
{
        int x,y;

        /* save cursor */
        x=outgetx();
        y=outgety();
        pmtmode1(mode);
        /* user types command on prompt line */
        getcmnd(buffer,outgetx());
        /* restore cursor */

        /* --- new ---
        outxy(x,y);
        ----- end comment out */
}


/* Update and print mode. */

pmtmode1(s)
char *s;
{
        int i;

        outxy(40,0);
        fmtsout(s,40);
        i=0;
        while (pmtln[i++]=*s++) {
                ;
        }
}


/* Print the file name on the prompt line. */

pmtfile1(s)
char *s;
{
        int i;

        outxy(25,0);
        if (*s==EOS) {
                fmtsout("no file",25);
        }
        else {
                fmtsout(s,25);
        }
        i=0;
        while (pmtfn[i++]=*s++) {
                ;
        }
}


/* Print the line number on the prompt line. */

pmtline1()
{
        outxy(0,0);
        fmtsout("line: ",0);
        putdec(bufln(),5);
}

/* Print column number of the cursor. */

pmtcol1(x)
int x;
{

        /* comment out all the following code if you do not
         * want column numbers to be drawn on the screen.
         * some people complain of too much flicker.
         */

        outxy(12,0);
        fmtsout("column: ",12);
        putdec(x,3);
}
```

```
/*
        RED general utilities -- small-C version

        Source:  red9.sc
        Version: August 27, 1982.

        Copyright (C) 1983 by Edward K. Ream
*/
```

```
/*      Return larger of two numbers.   */

max(a, b)
int a, b;
{
        if (a >= b) {
                return a;
        }
        else {
                return b;
        }
}


/*      Return smaller of two numbers.  */

min(a, b)
int a, b;
{
        if (a <= b) {
                return a;
        }
        else {
                return b;
        }
}


/*      Return the absolute value of a number.  */

abs(n)
int n;
{
        if (n < 0) {
                return -n;
        }
        else {
                return n;
        }
}


/*      Convert a character to lower case.      */

tolower(c)
char c;
{
        if ( (c >= 'A') & (c <= 'Z') ) {
                return c - 'A' + 'a';
        }
        else {
                return c;
        }
}

/*
        return: is first token in args a number ?
        return value of number in *val
*/

number(args,val) char *args; int *val;
{
char c;
        c=*args++;
        if ((c<'0')|(c>'9')) {
                return(NO);
        }
        *val=c-'0';
        while (c=*args++) {
                if ((c<'0')|(c>'9')) {
                        break;
                }
                *val=(*val*10)+c-'0';
        }
        return(YES);
}


/* Convert character buffer to numeric. */

ctoi(buf,index) char *buf; int index;
{
int k;
        while ( (buf[index]==' ') |
                (buf[index]==TAB) ) {
                index++;
        }
        k=0;
        while ((buf[index]>='0')&(buf[index]<='9')) {
                k=(k*10)+buf[index]-'0';
                index++;
        }
        return(k);
}


/*
        Put decimal integer n in field width >= w.
        Left justify the number in the field.
*/

putdec(n,w) int n,w;
{
char chars[10];
int i,nd;
        nd=itoc(n,chars,10);
        i=0;
        while (i<nd) {
                syscout(chars[i++]);
        }


        i=nd;
        while (i++<w) {
                syscout(' ');
        }
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 62)

```
/* Convert integer n to character string in str. */

itoc(n,str,size) int n; char *str; int size;
{
int absval;
int len;
int i,j,k;
        absval=abs(n);
        /* generate digits */
        str[0]=0;
        i=1;
        while (i<size) {
                str[i++]=(absval%10)+'0';
                absval=absval/10;
                if (absval==0) {
                        break;
                }
        }
        /* generate sign */
        if ((i<size)&(n<0)) {
                str[i++]='-';
        }
        len=i-1;
        /* reverse sign, digits */
        i--;
        j=0;
        while (j<i) {
                k=str[i];
                str[i]=str[j];
                str[j]=k;
                i--;
                j++;
        }
        return(len);
}


/* System error routine. */

syserr(s) char *s;
{
        pmtmess("system error: ",s);
}


/* User error routine. */

error(s) char *s;
{
        pmtmess("error: ",s);
}
```

End Listing One

Listing Two

```
#asm
; Part I of the Small-C Run-time Library
; Functions called from the code generators
;
; Source:  lib1
; Version: August 5, 1982
;

;
;Fetch a single byte from the address in HL and
; sign extend into HL
CCGCHAR: MOV     A,M
CCSXT:   MOV     L,A
         RLC
         SBB     A
         MOV     H,A
         RET
;Fetch a full 16-bit integer from the address in HL
CCGINT:  MOV     A,M
         INX     H
         MOV     H,M
         MOV     L,A
         RET
;Store a single byte from HL at the address in DE
CCPCHAR: MOV     A,L
         STAX    D
         RET
;Store a 16-bit integer in HL at the address in DE
CCPINT:  MOV     A,L
         STAX    D
         INX     D
         MOV     A,H
         STAX    D
         RET
;Inclusive "or" HL and DE into HL
CCOR:    MOV     A,L
         ORA     E
         MOV     L,A
         MOV     A,H
         ORA     D
         MOV     H,A
         RET
;Exclusive "or" HL and DE into HL
CCXOR:   MOV     A,L
         XRA     E
         MOV     L,A
         MOV     A,H
         XRA     D
         MOV     H,A
         RET
;"And" HL and DE into HL
CCAND:   MOV     A,L
         ANA     E
         MOV     L,A
         MOV     A,H
         ANA     D
         MOV     H,A
         RET
;Test if HL = DE and set HL = 1 if true else 0
CCEQ:    CALL    CCCMP
         RZ
         DCX     H
         RET
;Test if DE ~= HL
CCNE:    CALL    CCCMP
         RNZ
         DCX     H
         RET
;Test if DE > HL (signed)
CCGT:    XCHG
         CALL    CCCMP
         RC
         DCX     H
         RET
;Test if DE <= HL (signed)
CCLE:    CALL    CCCMP
         RZ
         RC
         DCX     H
         RET
;Test if DE >= HL (signed)
CCGE:    CALL    CCCMP
         RNC
         DCX     H
         RET
;Test if DE < HL (signed)
CCLT:    CALL    CCCMP
         RC
         DCX     H
         RET
;Common routine to perform a signed compare
; of DE and HL
;This routine performs DE - HL and sets the conditions:
;       Carry reflects sign of difference (set means DE < HL)
;       Zero/non-zero set according to equality.
CCCMP:   MOV     A,E
         SUB     L
         MOV     E,A
         MOV     A,D
         SBB     H
         LXI     H,1     ;preset true condition
         JM      CCCMP1
         ORA     E       ;"OR" resets carry
         RET
CCCMP1:  ORA     E
         STC             ;set carry to signal minus
         RET
;
;Test if DE >= HL (unsigned)
CCUGE:   CALL    CCUCMP
         RNC
         DCX     H
         RET
;
;Test if DE < HL (unsigned)
CCULT:   CALL    CCUCMP
         RC
         DCX     H
         RET
;
;Test if DE > HL (unsigned)
CCUGT:   XCHG
         CALL    CCUCMP
         RC
         DCX     H
         RET
;
;Test if DE <= HL (unsigned)
CCULE:   CALL    CCUCMP
         RZ
         RC
         DCX     H
         RET
;
;Common routine to perform unsigned compare
;carry set if DE < HL
;zero/nonzero set accordingly
CCUCMP:  MOV     A,D
         CMP     H
         JNZ     $+5
         MOV     A,E
         CMP     L
         LXI     H,1
         RET
;
;Shift DE arithmetically right by HL and return in HL
CCASR:   XCHG
         MOV     A,H
         RAL
         MOV     A,H
         RAR
```

```
        MOV     H,A
        MOV     A,L
        RAR
        MOV     L,A
        DCR     E
        JNZ     CCASR+1
        RET
;Shift DE arithmetically left by HL and return in HL
CCASL:  XCHG
        DAD     H
        DCR     E
        JNZ     CCASL+1
        RET
;Subtract HL from DE and return in HL
CCSUB:  MOV     A,E
        SUB     L
        MOV     L,A
        MOV     A,D
        SBB     H
        MOV     H,A
        RET
;Form the two's complement of HL
;
cmh:
CCNEG:  CALL    CCCOM
        INX     H
        RET
;Form the one's complement of HL
CCCOM:  MOV     A,H
        CMA
        MOV     H,A
        MOV     A,L
        CMA
        MOV     L,A
        RET

;Multiply DE by HL and return in HL
CCMULT: MOV     B,H
        MOV     C,L
        LXI     H,0
CCMULT1: MOV    A,C
        RRC
        JNC     $+4
        DAD     D
        XRA     A
        MOV     A,B
        RAR
        MOV     B,A
        MOV     A,C
        RAR
        MOV     C,A
        ORA     B
        RZ
        XRA     A
        MOV     A,E
        RAL
        MOV     E,A
        MOV     A,D
        RAL
        MOV     D,A
        ORA     E
        RZ
        JMP     CCMULT1
;Divide DE by HL and return quotient in HL, remainder in DE
CCDIV:  MOV     B,H
        MOV     C,L
        MOV     A,D
        XRA     B
        PUSH    PSW
        MOV     A,D
        ORA     A
        CM      CCDENEG
        MOV     A,B
        ORA     A
        CM      CCBCNEG
        MVI     A,16
        PUSH    PSW
        XCHG
        LXI     D,0
CCDIV1: DAD     H
        CALL    CCRDEL
        JZ      CCDIV2
        CALL    CCCMPBCDE
        JM      CCDIV2
        MOV     A,L
        ORI     1
        MOV     L,A
        MOV     A,E
        SUB     C
        MOV     E,A
        MOV     A,D
        SBB     B
        MOV     D,A
CCDIV2: POP     PSW
        DCR     A
        JZ      CCDIV3
        PUSH    PSW
        JMP     CCDIV1
CCDIV3: POP     PSW
        RP
        CALL    CCDENEG
        XCHG
        CALL    CCDENEG
        XCHG
        RET
CCDENEG: MOV    A,D
        CMA
        MOV     D,A
        MOV     A,E
        CMA
        MOV     E,A
        INX     D
        RET

CCBCNEG: MOV    A,B
        CMA
        MOV     B,A
        MOV     A,C
        CMA
        MOV     C,A
        INX     B
        RET
CCRDEL: MOV     A,E
        RAL
        MOV     E,A
        MOV     A,D
        RAL
        MOV     D,A
        ORA     E
        RET
CCCMPBCDE: MOV  A,E
        SUB     C
        MOV     A,D
        SBB     B
        RET
#endasm
```

```
#asm
; library for small-C -- Part II
;
; Source:  lib2
; Version: September 3, 1982
;
; Adapted from: CCC.ASM (C.CCC).  BDS C version 1.45,  11/22/81
; Original written by Leor Zolman
; Modification by Edward K. Ream
;


base:   equ     0       ;start of ram in system
bdos:   equ     base+5
tpa:    equ     base+100h
fcb:    equ     base+5ch
nfcbs:  equ     8       ;maximum # of files open at one time
tbuff:  equ     base+80h
origin: equ     tpa
exitad: equ     base    ;warm boot location

;
; define ASCII codes;
;

cr:     equ 0dh         ;carriage return
lf:     equ 0ah         ;linefeed
newlin: equ lf          ;newline
tab:    equ 9           ;tab
bs:     equ 08h         ;backspace
cntrlc: equ 3           ;control-C

;
; define BDOS call codes:
;

 errorv  equ 255         ;error value from BDOS

 conin:  equ 1           ;get a character from console
 conout: equ 2           ;write a character to console
 lstout: equ 5           ;write a character to list device
 dconio: equ 6           ;direct console I/O (only for CP/M 2.0)
 pstrng: equ 9           ;print string (terminated by '$')
 getlin: equ 10          ;get buffered line from console
 cstat:  equ 11          ;get console status
 select: equ 14          ;select disk
 openc:  equ 15          ;open a file
 closec: equ 16          ;close a file
 delc:   equ 19          ;delete a file
 reads:  equ 20          ;read a sector (sequential)
 writs:  equ 21          ;write a sector (sequential)
 creatc: equ 22          ;make a file
 renc:   equ 23          ;rename file
 sdma:   equ 26          ;set dma
 readr:  equ 33          ;read random sector
 writr:  equ 34          ;write random sector
 cfsizc: equ 35          ;compute file size
 srrecc: equ 36          ;set random record

 ;
 ; Define global varialbes used by the library
 ;

 args:   ds      14      ;"arghak" puts args here.
 arg1    equ     args
 arg2    equ     args+2
 arg3    equ     args+4
 arg4    equ     args+6
 arg5    equ     args+8
 arg6    equ     args+10
 arg7    equ     args+12

 iohack: ds      6       ;room for I/O subroutines for use
                         ;by "inp" and "outp" library routines
                         ;(initialized by init)

 rseed   ds      8       ;seed for random number routines
                         ;(initialized by init)

 tmp:    ds      1       ; misc. garbage space
 tmp1:   ds      2       ;
 tmp2:   ds      2       ;
 tmp2a:  ds      2       ;

 ungetl: ds      1       ;where characters are "ungotten"
 lastc:  ds      1       ;last char typed

 allocp: ds      2       ;storage allocation pointers
 alocmx: ds      2       ;(initialized by init)
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 62)

```
;
; The fcb table (fcbt): 36 bytes per file control block
;

fcbt:   ds      36*nfcbs        ;reserve room for fcb's

;
; The fd table: one byte per file specifying r/w/open as follows:
;       bit 0 is high if open, low if closed
;       bit 1 is high if open for read
;       bit 2 is high if open for write
; (both b1 and b2 may be high)
;

fdt:    ds      nfcbs   ;one byte per fcb

;
; The command line is copied here by init:
;

comlin: ds      131     ;copy of the command line

;
; This is where "init" places the array of argument pointers:
;
; the "argv" parameter points 2 bytes before arglst.
; thus, up to 30 parameters may be passed to main().

arglst: ds      60


;
; Because small-C pushes arguments in the order in which
; they appear in a program, it is not possible to know where
; the first argument is unless the number of arguments is
; known.
;
; The following three routines move arguments from the
; stack into the argument area, depending on the number
; of arguments to the function.
;
get3args:       lxi     h,4     ;point at arg 3
                dad     sp

                push    h       ;get it
                mov     a,m
                inx     h
                mov     h,m
                mov     l,a
                shld    arg3
                pop     h
                inx     h       ;point at arg 2
                inx     h
                jmp     get2next

get2args:       lxi     h,4     ;point at arg 2
                dad     sp

get2next:       push    h
                mov     a,m     ;get arg 2 from stack
                inx     h
                mov     h,m
                mov     l,a
                shld    arg2    ;store it
                pop     h
                inx     h
                inx     h
                jmp     get1next

get1arg:        lxi     h,4     ;point at arg 1
                dad     sp

get1next:       mov     a,m     ;get arg 1 from stack
                inx     h
                mov     h,m
                mov     l,a
                shld    arg1    ;store it
                ret


;
; These routines get the indicated parameter into both
; the a and hl regs, assuming that the correct call to
; either get1arg, get2args or get3args has been made.
;

ma1toh:         lhld    arg1
                mov     a,l
                ret

ma2toh:         lhld    arg2
                mov     a,l
                ret

ma3toh:         lhld    arg3
                mov     a,l
                ret

;
; This routine is called first to do argc & argv processing
; and some odds and ends initializations:
; --- push argc and argv in reverse order from BDS C ----
;
ccgo:
init:   pop     h       ;store return address
        shld    tmp2    ; somewhere safe for the time being

;--- push later ---
;       lxi     h,arglst-2      ;set the "argv" that the
;       push    h               ; main program will get.

;
; Initialize storage allocation pointers:
;
;--- you  can  add these if you modify the small-C compiler ---
;--- so that it defines the label freram to point after the ---
;--- end of the externals                                   ---
;
;       lxi     h,freram ;get address after end of externals
;       shld    allocp  ;store at allocation pointer for sbrk()
;       lxi     h,1000  ;default safety space between stack and
;       shld    alocmx  ; highest allocatable address in memory
                        ; (for use by "sbrk".).

;
; Initialize random seed:
;
        lxi     h,59dch ;let's stick something wierd into the
        shld    rseed   ;first 16 bits of the random-number seed

;
;Initialize I/O hack locations:
;

        mvi     a,0dbh          ;"in" op, for "in xx; ret"
        sta     iohack
        mvi     a,0d3h          ;"out" op for "out xx; ret"
        sta     iohack+3
        mvi     a,0c9h          ;"ret" for above subroutines
        sta     iohack+2        ;the port number is filled in
        sta     iohack+5        ;by "inp" and "outp"


        mvi     c,11    ;interrogate console status to see if
        call    bdos    ;there is a stray character.

        ora     a       ;used to be `ani 1'... better for
        nop             ;for some CP/M like systems

        jz      initzz
        mvi     c,1     ;if input present, clear it
        call    bdos

initzz: lxi     h,tbuff         ;if arguments given, process them.
        lxi     d,comlin        ;get ready to copy command line
        mov     b,m             ;first get length of it
        inx     h
        mov     a,b
        ora     a       ;if no arguments, don't parse for argv
        jnz     initl
        lxi     d,1     ;set argc to 1 in such a case.
        jmp     i5

initl:  mov     a,m     ;ok, there are arguments. parse...
        stax    d       ;first copy command line to comlin
        inx     h
        inx     d
        dcr     b
        jnz     initl
        xra     a       ;place zero following line
        stax    d

        lxi     h,comlin        ;now compute pointers to each arg
        lxi     d,1             ;arg count
        lxi     b,arglst        ;where pointers will all go
        xra     a               ;clear "in a string" flag
        sta     tmp1
i2:     mov     a,m     ;between args...
        inx     h
        cpi     ' '
        jz      i2
        ora     a
        jz      i5      ;if null byte, done with list
        cpi     '"'
        jnz     i2a     ;quote?
        sta     tmp1    ;yes. set "in a string" flag
        jmp     i2b

i2a:    dcx     h
i2b:    mov     a,l     ;ok, HL is a pointer to the start
        stax    b       ;of an arg string. store it.
        inx     b
        mov     a,h
        stax    b
        inx     b
        inx     d       ;bump arg count
i3:     mov     a,m
        inx     h       ;pass over text of this arg
        ora     a       ;if at end, all done
        jz      i5
        push    b       ;if tmp1 set, in a string
        mov     b,a     ; (so we have to ignore spaces)
        lda     tmp1
        ora     a
        mov     a,b
        pop     b
        jz      i3a
        cpi     '"'     ;we are in a string.
        jnz     i3      ;check for terminating quote
        xra     a       ;if found, reset "in string" flag
        sta     tmp1
        dcx     h
        mov     m,a     ;and stick a zero byte after the string
        inx     h       ;and go on to next arg
i3a:    cpi     ' '     ;now find the space between args
        jnz     i3
        dcx     h       ;found it. stick in a zero byte
        mvi     m,0
        inx     h
        jmp     i2      ;and go on to next arg
```

```
i5:     push    d       ;all done finding args. Set argc.
        lxi     h,arglst-2      ;--- now push argv ---
        push    h

        mvi     b,nfcbs ;now initialize all the file info
        lxi     h,fdt   ;(just zero the fd table)
i6:     mvi     m,0
        inx     h
        dcr     b
        jnz     i6

        xra     a
        sta     ungetl  ;clear the push-back byte
        sta     lastc   ;and last character byte

        lhld    tmp2
        pchl            ;all done initializing.
;
; General purpose error value return routine:
;

error:  lxi     h,-1    ;general error handler...just
        ret             ;returns -1 in HL

;
; Here are file I/O handling routines, only needed under CP/M:
;


;
; Close any open files and reboot:
;

qzexit:
exit:
        mvi     a,7+nfcbs       ;start with largest possible fd
exit1:  push    psw             ;and scan all fd's for open files
        call    fgfd            ;is file whose fd is in A open?
        jc      exit2           ;if not, go on to next fd
        mov     l,a             ;else close the associated file
        mvi     h,0
        push    h
        call    close
        pop     h
exit2:  pop     psw
        dcr     a               ;and go on to next one
        cpi     7
        jnz     exit1

        jmp     exitad          ;done closing; now reboot CP/M

;
; Close the file whose fd is 1st arg:
;

qzclose:
close:
        call    getlarg ;--- new ---
        call    setdma  ;library function just jumps here.
        call    maltoh  ;get fd in A
        call    fgfd    ;see if it is open
        jc      error   ;if not, complain
        mov     a,m
        ani     4
        jz      close2  ;the file isn't open for write
        push    h       ;save fd table entry addr
        call    maltoh  ;---was ma2toh --- move arg1 to A
        push    b
        call    fgfcb   ;get the appropriate fcb address
        xchg            ;put it in DE
        mvi     c,16    ;get BDOS function # for close
        call    bdos    ;and do it!
        pop     b
        pop     h
close2: mvi     m,0     ;close logically
        cpi     255     ;if 255 comes back, we got problems
        lxi     h,0
        rnz             ;return 0 if OK
        dcx     h       ;return -1 on error
        ret
;
; Determine status of file whose fd is in A...if the file
; is not open, return C flag set, else clear C flag:

fgfd:   call    setdma
        mov     d,a
        sui     8
        rc              ;if fd < 8, error
        cpi     nfcbs
        cmc             ;don't allow too big an fd either
        rc
        push    d
        mov     e,a     ;OK, we have a value in range. Now
        mvi     d,0     ;  see if the file is open or not
        lxi     h,fdt
        dad     d
        mov     a,m
        ani     1       ;bit 0 is high if file is open
        stc
        pop     d
        mov     a,d
        rz              ;return C set if not open
        cmc
        ret             ;else reset C and return
;
; Set up a CP/M file control block at HL with the file whose
; simple null-terminated name is pointed to by DE:
; Format for filename must be: "[white space][d:]filename.ext"
;

setfcb:
        call    setdma  ;set up an fcb at HL for filename at DE
        push    b
        call    igwsp   ;ignore blanks and tabs
        mvi     b,8
        push    h
        inx     d
        ldax    d
        dcx     d
        cpi     ':'     ;default disk byte value is 0
        mvi     a,0     ; (for currently logged disk)
        jnz     setf1
        ldax    d       ;oh oh...we have a disk designator
        call    mapuc   ;make it upper case
        sui     '@'     ;and fudge it a bit
        inx     d
        inx     d
setf1:  mov     m,a
        inx     h
        call    patchnm ;now set filename and pad with blanks
        ldax    d
        cpi     '.'     ;and if an extension is given,
        jnz     setfcb2
        inx     d
setfcb2 mvi     b,3     ;set the extension and pad with blanks
        call    setnm
        xra     a       ;and zero the appropriate fields of the fcb
        mov     m,a
        lxi     d,20
        dad     d
        mov     m,a
        inx     h
;
        mov     m,a     ;zero the random record bytes
        inx     h
        mov     m,a
        inx     h
        mov     m,a
        pop     d
        pop     b
        ret

patchnm:
        call    setnm   ;another patch from "vsetfcb"
        jmp     setnm3

;
; This routine copes up to B characters from memory at DE to
; memory at HL and pads with blanks on the right:
;

setnm:  push    b
setnm1: ldax    d
        cpi     '*'     ;wild card?
        mvi     a,'?'   ;if so, pad with ? characters
        jz      pad2

setnm2: ldax    d
        call    legfc   ;next char legal filename char?
        jc      pad     ;if not, go pad for total of B characters
        mov     m,a     ;else store
        inx     h
        inx     d
        dcr     b
        jnz     setnm1  ;and go for more if B not yet zero
        pop     b
setnm3: ldax    d       ;skip rest of filename if B chars already found
        call    legfc
        rc
        inx     d
        jmp     setnm3

pad:    mvi     a,' '   ;pad with B blanks
pad2:   mov     m,a     ;pad with B instances of char in A
        inx     h
        dcr     b
        jnz     pad2
        pop     b
        ret

;
; Test if char in A is lega' character to be in a filename:
;

legfc:  call    mapuc
        cpi     '.'     ; '.' is illegal in a filename or extension
        stc
        rz
        cpi     ':'     ;so is ':'
        stc
        rz
        cpi     7fh     ;delete is no good
        stc
        rz
        cpi     '!'     ;if less than exclamation pt, not legal char
        ret             ;else good enough

;
; Map character in A to upper case if it is lower case:
;


mapuc:  cpi     'a'
        rc
        cpi     'z'+1
        rnc
        sui     32      ;if lower case, map to upper
        ret

;
; Ignore blanks and tabs at text pointed to by DE:
;
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 62)

```
igwsp:  dcx     d
igwsp1: inx     d
        ldax    d
        cpi     ' '
        jz      igwsp1
        cpi     9
        jz      igwsp1
        ret
;
; This routine does one of two things, depending
; on the value passed in A.
;
; If A is zero, then it finds a free file slot
;  (if possible), else returns C set.
;
; If A is non-zero, then it returns the address
; of the fcb corresponding to an open file whose
; fd happens to be the value in A, or C set if there
; is no file associated with fd.
;

fgfcb:  push    b
        call    setdma
        ora     a       ;look for free slot?
        mov     c,a
        jnz     fgfc2   ;if not, go away
        mvi     b,nfcbs ;yes. do it...
        lxi     d,fdt
        lxi     h,fcbt
        mvi     c,8
fgfc1:  ldax    d
        ani     1
        mov     a,c
        jnz     fgfc1a  ;found free slot?
        pop     b       ;yes. all done.
        ret
fgfc1a: push    d
        lxi     d,36    ;fcb length to accommodate random I/O
        dad     d
        pop     d
        inx     d
        inr     c
        dcr     b
        jnz     fgfc1
fgfc1b: stc
        pop     b
        ret             ;return C if no more free slots
fgfc2:  call    fgfd    ;compute fcb address for fd in A:
        jc      fgfc1b  ;return C if file isn't open

        sui     8
        mov     l,a     ;put (fd-8) in HL
        mvi     h,0
        dad     h       ;double it
        dad     h       ;4*a
        mov     d,h     ;save 4*a in DE
        mov     e,l
        dad     h       ;8*a
        dad     h       ;16*a
        dad     h       ;32*a
        dad     d       ;36*a
        xchg            ;put 36*a in DE
        lxi     h,fcbt  ;add to base of table
        dad     d       ;result in HL
        mov     a,c     ;and return original fd in A
        pop     b
        ret
setdma: push    d       ;just a preventative measure,
        push    b       ;since the default I/O buffer
        push    psw     ;tends to magically change
        push    h       ;around by itself when left
        mvi     c,26    ;in CP/M's hands !!
        lxi     d,tbuff
        call    bdos
        pop     h
        pop     psw
        pop     b
        pop     d
        ret
#endasm
```

```
#asm
; File I/O library for small-C -- Part I
;
; Source:  fiolib1
; Version: September 3, 1982
;
; Adapted from deff2.csm.  BDS C version 1.46,  3/22/82
; Original by Leor Zolman
; Modifications by Edward K. Ream
;
;
; Functions appearing in this file:
;
;       getchar kbhit   ungetch putchar putch   gets
;       setfcb  read    write   open    close   creat
;       unlink  seek    tell    rename
;       fabort  fcbaddr exit    bdos    bios
;

qzgetchar:
getchar:
        lda     ungetl  ;any character pushed back?
        ora     a
        mov     l,a
        jz      gch2
        xra     a       ;yes. return it and clear the pushback
        sta     ungetl  ;byte in C.CCC.
        mvi     h,0
        ret

gch2:   push    b
        mvi     c,conin
        call    bdos
        pop     b
        cpi     cntrlc  ;control-C ?
        jz      base    ;if so, reboot.
        cpi     1ah     ;control-Z ?
        lxi     h,-1    ;if so, return -1.
        rz
        mov     l,a
        cpi     cr      ;carriage return?
        jnz     gch3
        push    b
        mvi     c,conout        ;if so, also echo linefeed
        mvi     e,lf
        call    bdos
        pop     b
        mvi     l,newlin        ;and return newline (linefeed)..

gch3:   mvi     h,0
        ret


qzkbhit:
kbhit:
        lda     ungetl  ;any character ungotten?
        mvi     h,0
        mov     l,a
        ora     a
        rnz             ;if so, return true

        push    b
        mvi     c,cstat ;else interrogate console status
        call    bdos
        pop     b
        ora     a       ;0 returned by BDOS if no character ready
        lxi     h,0
        rz              ;return 0 in HL if no character ready
        inr     l       ;otherwise return 1 in HL
        ret


qzungetch:
ungetch:
        call    getlarg ;--- new ---
        lda     ungetl
        mov     l,a
        push    h
        call    maltoh  ;--- was ma2toh ---
        sta     ungetl
        pop     h
        mvi     h,0
        ret

;this is a new function

qzputs:
puts:
        call    getlarg
        call    maltoh
        push    b
        call    puts1
        pop     b
        ret

puts1:  mov     a,m
        cpi     0
        rz
        push    h               ;use putchar's interrupt logic
        mov     l,a
        push    h
        call    putchar
        pop     h
        pop     h
        inx     h
        jmp     puts1


qzputchar:
putchar:
        call    getlarg ;--- new ---
        call    maltoh  ;get character in A
        push    b
        mvi     c,conout
        cpi     newlin  ;newline?
        jnz     put1    ;if not, just go put out the character
        mvi     e,cr    ;else...put out CR-LF
        call    bdos
        mvi     c,conout
        mvi     a,lf

put1:   mov     e,a
        call    bdos

put2:   mvi     c,cstat ;now, is input present at the console?
        call    bdos
        ora     a
        jnz     put3
        pop     b       ;no...all done.
        ret
put3:   mvi     c,conin ;yes. sample it (this will always echo the
        call    bdos    ;       character to the screen, alas)
        cpi     cntrlc  ;is it control-C?
        jz      base    ;if so, abort and reboot
        pop     b       ;else ignore it.
        ret


qzputch:
putch:
        call    getlarg ;--- new ---
        call    maltoh
```

```
        push    b
        mvi     c,conout
        mov     e,a
        cpi     newlin
        jnz     putch1  ;if not newline, just put it out
        mvi     e,cr    ;else put out CR-LF
        call    bdos
        mvi     c,conout
        mvi     e,lf
putch1: call    bdos
        pop     b
        ret


qzgets:
gets:
        call    get1arg ;--- new ---
        call    maltoh  ;get destination address
        push    b       ;save BC
        push    h
        push    h
        lxi     h,-150  ;use space below stack for reading line
        dad     sp
        push    h       ;save buffer address
        mvi     m,88h   ;Allow a max of about 135 characters
        mvi     c,getlin
        xchg            ;put buffer addr in DE
        call    bdos    ;get the input line
        mvi     c,conout
        mvi     e,lf    ;put out a LF
        call    bdos
        pop     h       ;get back buffer address
        inx     h       ;point to returned char count
        mov     b,m     ;set B equal to char count
        inx     h       ;HL points to first char of line
        pop     d       ;DE points to start destination area
copy1:  mov     a,b     ;copy line to start of buffer
        ora     a
        jz      gets2
        mov     a,m
        stax    d
        inx     h
        inx     d
        dcr     b
        jmp     copy1

gets2:  xra     a       ;store terminating null
        stax    d
        pop     h       ;return buffer address in HL
        pop     b
        ret


qzsetfcb:
        call    get2args        ;--- was arghak ---
        push    b
        lhld    arg2    ;get pointer to name text
igsp:   mov     a,m
        inx     h
        cpi     ' '
        jz      igsp
        cpi     tab
        jz      igsp
        dcx     h
        xchg            ;set DE pointing to 1st non-space char
        lhld    arg1    ;get --> fcb area
        call    setfcb  ; do it
        lxi     h,0     ;all OK.
        pop     b
        ret


qzread:
read:
        call    get3args        ;--- was arghak ---
        lda     arg1
        call    fgfd
        jc      error   ;error if illegal fd
        mov     a,m
        ani     2       ;open for read?
        jz      error   ;error if not
        push    b
        lda     arg1
        call    fgfcb
        shld    tmp2    ;tmp2 will hold dma addr
        lxi     h,0     ;count of # of successful sectors read
        shld    tmp2a   ; will be kept at tmp2a
read2:  lhld    arg3    ;done?
        mov     a,h
        ora     l
        jz      read4

read2a: lhld    arg2    ;else read another sector
        xchg            ;DE is dma addr
        mvi     c,sdma
        call    bdos    ;set DMA
        lhld    tmp2
        xchg            ;DE is fcb addr
        mvi     c,reads
        push    d       ;save de so we can fudge nr field if
        call    bdos    ;we stop reading on extent boundary...
        pop     d               ; CP/M sucks!
        cpi     2
        pop     b
        jz      error   ;if error, abort
        push    b
        cpi     1
        jnz     read6   ;EOF?

read3:  lxi     h,32    ;yes. are we on extent boundary?
        dad     d       ;if so, adjust for CP/M's stupidity here
        mov     a,m     ;by turning an 80h sector count into 00h.
        cpi     80h
        jnz     read4
        mvi     m,0     ;yes. reset nr to 0...
read4:  lhld    tmp2a
read5:  pop     b
        ret

read6:  lhld    arg3
        dcx     h
        shld    arg3
        lhld    arg2
        lxi     d,128
        dad     d
        shld    arg2
        lhld    tmp2a
        inx     h
        shld    tmp2a
        jmp     read2


qzwrite:
write:
        call    get3args        ;--- was arghak ---
        lda     arg1
        call    fgfd
        jc      error
        mov     a,m
        ani     4
        jz      error
        push    b
        lda     arg1
        call    fgfcb
        shld    tmp2
        lxi     h,0
        shld    tmp2a
        lxi     d,tbuff         ;80 for normal CP/M, else 4280
        mvi     c,sdma
        call    bdos

writ1:  lhld    arg3    ;done yet?
        mov     a,h
        ora     l
        lhld    tmp2a   ;if so, return count
        jz      writ3
        lhld    arg2    ;else copy next 128 bytes down to tbuff
        lxi     d,tbuff ;80 for normal CP/M, else 4280
        mvi     b,128
writ2:  mov     a,m
        stax    d
        inx     h
        inx     d
        dcr     b
        jnz     writ2
        shld    arg2    ;save -> to next 128 bytes
        lhld    tmp2    ;get addr of fcb
        xchg
        mvi     c,writs ;go write
        call    bdos
        ora     a       ;error?
        lhld    tmp2a   ;if so, return # of successfully written
        jnz     writ3   ;  sectors.

        inx     h       ; else bump successful sector count,
        shld    tmp2a
        lhld    arg3    ; debump countdown,
        dcx     h
        shld    arg3
        jmp     writ1   ; and go try next sector
writ3:  pop     b
        ret


qzopen:
open:
        call    get2args        ;--- was arghak ---
        xra     a
        call    fgfcb   ;any fcb's free?
        jc      error   ;if not, error
        sta     tmp
        xchg
        lhld    arg1
        xchg
        push    b
        call    setfcb
        mvi     c,openc
        call    bdos
        cpi     errorv  ;successful open?
        pop     b
        jz      error   ;if not, error
        lda     tmp
        call    fgfd    ;get HL pointing to fd table entry
        lda     arg2
        ora     a       ;open for read?
        mvi     d,3
        jz      open1
        dcr     a
        mvi     d,5
        jz      open1   ;write?
        dcr     a
        jnz     error   ;else must be both or bad mode.
        mvi     d,7
open1:  mov     m,d
        lda     tmp
        mov     l,a
        mvi     h,0
        ret


qzcreat:
creat:
        call    get1arg ;--- was arghak ---
        lhld    arg1
        push    b
```

(Continued on next page)

RED - Listing

(Listing continued, text begins on page 62)

```
        push    h
        call    unlink  ;erase any old versions of file
        pop     d
        mvi     c,creatc
        lxi     d,fcb
        call    bdos
        cpi     errorv
        pop     b
        jz      error

        lhld    arg1    ;--- push these in reverse order ---
        push    h
        lxi     h,2     ;open for read/write
        push    h

        call    open
        pop     d
        pop     d
        ret
qzunlink:
unlink:
        call    get1arg ;--- new ---
        call    maltoh
        push    b
        xchg
        lxi     h,fcb
        call    setfcb
        mvi     c,delc
        call    bdos
        lxi     h,0
        pop     b
        ret
qzseek:
seek:
        call    get3args        ;--- was arghak ---
        lda     arg1
        call    fgfcb
        jc      error   ;error if file not open
        push    b
        push    h       ;save fcb address

        lhld    arg1
        push    h
        call    tell    ;get r/w pointer position for the file
        pop     d

        xchg            ;put present pos in DE
        lda     arg3
        lhld    arg2    ;get offset in HL
        ora     a       ;absolute offset?
        jz      seek2   ;if so, offset is new position
        dad     d       ;else add offset to current position
seek2:  mov     a,l     ;convert to extent and sector values
        rlc
        mov     a,h
        ral
        ani     7fh
        sta     tmp
        xthl
        lxi     d,12
        push    h
        dad     d
        cmp     m       ;jumping over extent boundary?
        jz      seek5
        xthl            ;yes.
        xchg
        mvi     c,closec        ;close old extent
        push    d
        call    bdos
        pop     d
        pop     h
        cpi     errorv
        jnz     seek4
seek3:  pop     d
        pop     b
        jmp     error

seek4:  lda     tmp
        mov     m,a
        push    d
        mvi     c,openc ;and open new one.
        call    bdos
seek5:  pop     d
        cpi     errorv
        jz      seek3
        lxi     h,32    ;and set nr field
        dad     d
        pop     d
        mov     a,e
        ani     7fh
        mov     m,a
        xchg            ;return new sector # in HL
        pop     b
        ret
qztell:
tell:
        call    get1arg ;--- new ---
        call    maltoh  ;get fd value in A
        call    fgfcb
        jc      error

        push    b
        lxi     d,12
        dad     d
        mov     b,m     ;put extent # in B
        lxi     d,20
        dad     d
        mov     c,m     ;put sector # in C
        xra     a       ;rotate extent right one bit
                        ;old b0 --> Carry
        mov     a,b
        rar
        mov     h,a     ;rotated value becomes high byte of
                        ;tell position
        mvi     a,0     ;rotate b0 of extent into A
        rar
        mov     b,a     ;save rotated extent number in B
        add     c       ;add rotated extent number to sector #
        mov     l,a     ;and result becomes low byte of
                        ;tell position
        mov     a,c     ;if both rotated extent # and sector #
                        ;has bit 7 hi,
        ana     b       ;then the sum had an overflow, so...
        jp      tell2
        inr     h       ;bump position number by 256
tell2:  pop     b       ;and all done.
        ret


qzrename:
rename:
        call    get2args        ;--- was arghak ---
        push    b
renam:  lhld    arg1
        xchg
        lxi     h,wfcb
        call    setfcb
        lhld    arg2
        xchg
        lxi     h,wfcb+16
        call    setfcb
        lxi     d,wfcb
        mvi     c,renc
        call    bdos
        pop     b

        cpi     errorv
        jz      error
        lxi     h,0
        ret
wfcb:   ds 53


qzfabort:
fabort:
        call    get1arg ;--- new ---
        call    maltoh
        call    fgfd
        jc      error
        mvi     m,0     ;clear entry in fd table
        lxi     h,0
        ret


qzfcbaddr:
fcbaddr:
        call    get1arg ;--- new ---
        call    maltoh
        call    fgfd    ;is it an open file?
        jc      error
        call    maltoh
        call    fgfcb   ;get fcb addr in HL
        ret


bdos    equ     5
qzbdos:
        call    get2args        ;--- was arghak ---
        push    b
        lda     arg1    ;get C value
        mov     c,a
        lhld    arg2    ;get DE value
        xchg            ;put in DE
        call    bdos    ;make the bdos call
        pop     b
        ret             ;and return to caller


qzbios:
bios:
        call    get2args        ;--- was arghak ---
        push    b
        lhld    base+1  ;get addr of jump table + 3
        dcx     h       ;set to addr of first jump
        dcx     h
        dcx     h
        lda     arg1    ;get function number (1-85)
        mov     b,a     ;multiply by 3
        add     a
```

```
        add     b
        mov     e,a     ;put in DE
        mvi     d,0
        dad     d       ;add to base of jump table
        push    h       ;and save for later
        lhld    arg2    ;get value to be put in BC
        mov     b,h     ;and put it there
        mov     c,l
        lxi     h,retadd        ;where call to bios will return to
        xthl            ;get address of vector in HL
        pchl            ;and go to it...
retadd: mov     l,a     ;all done. now put return value in HL
        mvi     h,0
        pop     b
        ret             ;and return to caller
#endasm
```

```
#asm
; Extra small-C run time library
;
; Source:  xlib
; Version: August 29, 1982
;
; Adapted from deff2a.csm.  BDS C version 1.46,  3/22/82
; Original by Leor Zolman
; Modified by Edward K. Ream
;

qzsetjmp:
        call    getlarg ;--- new ---
        call    maltoh

        mov     m,c     ;save BC (not really needed in small-C)
        inx     h
        mov     m,b
        inx     h

        xchg            ;save SP
        lxi     h,0
        dad     sp
        xchg
        mov     m,e
        inx     h
        mov     m,d
        inx     h

        pop     d       ;save return address
        push    d
        mov     m,e
        inx     h
        mov     m,d
        lxi     h,0     ;and return 0
        ret

;longjmp(buffer, return_value)
;
;       unlike the BDS C version of this routine,
;       the return_value is REQUIRED
;

qzlongjmp:
        call    get2args        ;--- new ---
        call    maltoh  ;get buffer address

        mov     c,m     ;restore BC
        inx     h
        mov     b,m
        inx     h

        mov     e,m     ;restore SP...first put it in DE
        inx     h
        mov     d,m
        inx     h

        shld    temp    ;save pointer to return address
        call    ma2toh  ;get return value

        xchg            ;put return val in DE, old SP in HL
        sphl            ;restore SP with old value

        pop     h       ;pop return address off stack
        lhld    temp    ;get back ptr to return address

        mov     a,m
        inx     h
        mov     h,m
        mov     l,a     ;HL holds new return address

        xchg            ;ret addr in DE, return value in HL
        push    d       ;push return address on stack
        ret             ;and return...

temp:   ds 2
#endasm
```

End Listing Two

CP/M EXCHANGE

by Robert Blum

A lot of space has been devoted to discussing CP/M Plus's new architecture and how it should improve program execution speed relative to I/O operations. Until recently it was all speculation, but benchmark comparisons are now becoming available [see this month's Clinic, page 10]. Most of the reports I have seen point out that some improvement can be expected straight out of the box. I think the real story will begin to unfold as it gains more extensive use in the hacker community.

Quickening the Pace

Carl Bochert of San Jose, California called me several weeks ago to share his experiences with CP/M Plus. His system is configured with 256K of memory to allow plenty of work space for directory hash tables and disk buffers. To benchmark his system, Carl used RMAC, DR's relocating assembler, to assemble several sample programs. He experienced approximately a 17% reduction in runtime with the standard CP/M Plus system.

Carl also indicated that he made a few patches which produced even better results, but for the moment will reserve comment on their content.

CP/M Plus Memory Management

Last month I reviewed two memory management techniques that can be used to expand system memory beyond the normal 64K address space limitation of most 8-bit processors. In studying the following CP/M Plus memory models I think it will become more clear to you how all the pieces fit together.

To fully implement CP/M Plus requires at least 96K of memory divided into two switchable banks and one fixed segment. However, a typical system will probably go well beyond the minimum to at least 160K divided into three 48K switchable banks and one fixed 16K segment. I use 48K for the switchable area because most memory boards produced today have facilities that work properly with MP/M, which requires this segment size.

Examining a banked CP/M Plus memory model, Figure 1 (below), reveals the complexities that are faced when bringing the system up. The top portion of memory is called "common memory," which is always enabled and addressable. The operating system is divided into two modules: the resident portion, which resides in common memory, and the banked portion, which resides just below common memory in bank 0. All remaining memory in bank 0 is split for use by directory hash tables and data buffers. It is not a requirement for bank 0 to begin at location 0. When the system is combined with the system generation utility, GENCPM, starting addresses can be entered for each of the operating system modules and data storage tables.

Bank 1 is the transient program bank and must begin at location 0. All memory within bank 1 must be contiguous to the top of banked memory. Since most of the operating system is in a bank of memory dedicated to operating system use, it is entirely possible to achieve a TPA size of 62K – quite an increase over 56K, which has come to be the standard for CP/M 2.2.

Any remaining banked memory will

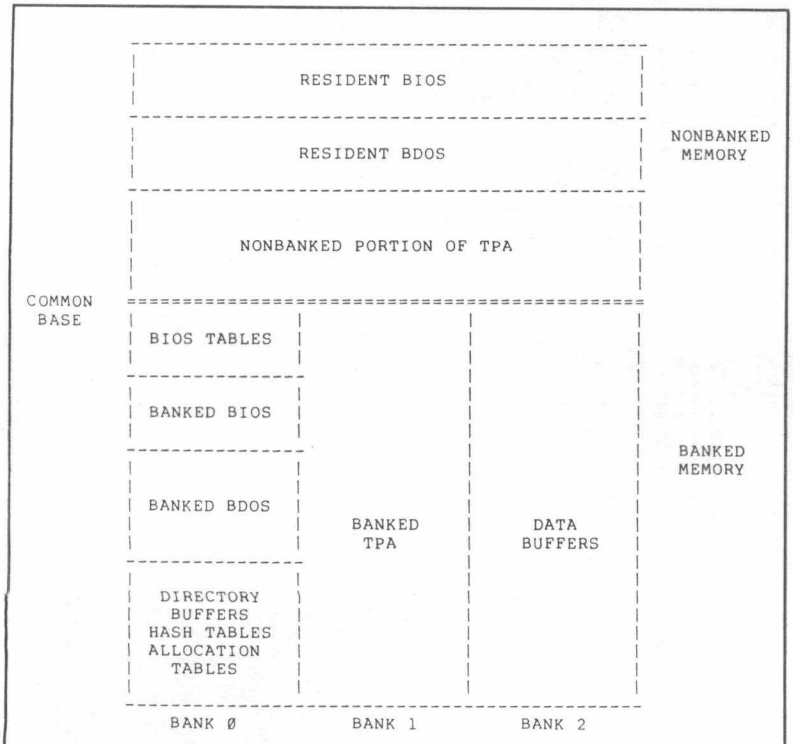


Figure 1.
Typical Nonbanked CP/M Plus System Model

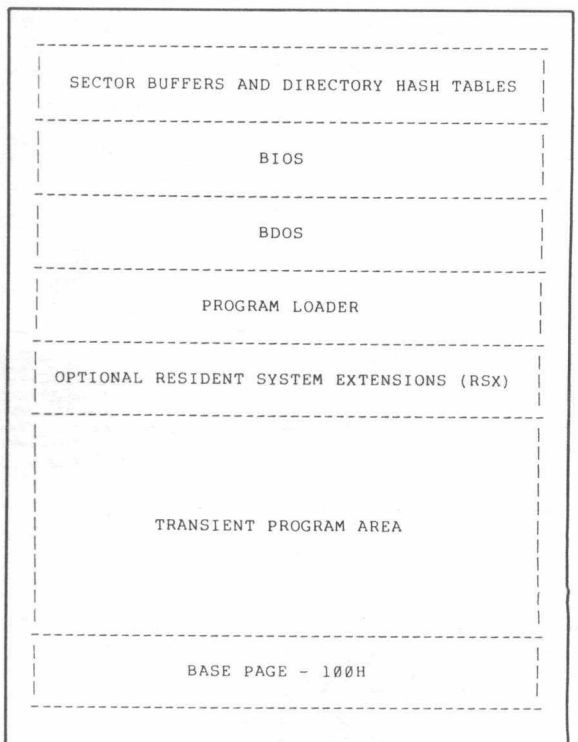


Figure 2.
Typical Banked CP/M Plus System Model

be used by CP/M Plus to maintain a cache of deblocking buffers and directory records. A very efficient Least Recently Used (LRU) buffering scheme is employed to manage these buffer pools. When buffer space is exhausted, the buffer that is least used is overwritten by incoming data and retrieved again later as needed.

The memory model for a nonbanked CP/M Plus, Figure 2 (page 98), is practically identical to CP/M 2.2 with the exception that at least an additional 2K will be used by the operating system. The resulting TPA of approximately 54K may be too small for some applications and must be seriously considered.

Setting Sail by the North Star

Ed McMurray of Columbia, South Carolina recently wrote asking some questions about Lifeboat's implementation of CP/M 2.2 for the North Star computer. His letter brought back a flood of fond memories. More years ago than I care to mention, I had finally convinced myself that I just had to have a floppy disk system to replace my TDL cassette tape interface. At that time only a few disk systems were available for the S-100 bus; and after looking around for the best price, I put down what seemed to be a small fortune to take home a North Star single-density controller board and a single Shugart SA400 disk drive. Obviously, everything was in kit form. After correcting a few solder bridges and establishing a suitable software interface to my I/O boards, I was on the air in grand style. My appetite had been whetted; soon one drive grew to two; double-density controller boards became available, and obviously I had to have one. For some unknown reason, I still didn't have enough storage. I couldn't have been happier when the UPS man brought my new 8-inch drives and controller. And now I keep hearing about these things called hard disks. Let's see – what excuse can I use to justify that new 40Meg drive?

Ed's question had to do with connecting different manufacturers' disk drives to the same controller and training the BIOS to step each one at a different rate. To insure that I had up-to-date information I made a call to Lifeboat to get their guidance. I was told that, in release 2.23A of their CP/M, a utility program, SETCPM, must be used to set the operational characteristics for each drive. The options that can be selected are fast or slow stepping rates and whether the drive is capable of single- or double-sided operation. In all cases double-density recording is assumed.

One alternative solution is to use the public domain North Star BIOS contained on volume 82 available from CPMUG. A brief review of the documentation for this program specifically points out that drives with differing capabilities (step rates, number of tracks, etc.) can be used on the same system.

Another Byte for Apple

The CP/M Users Group (CPMUG) is now offering their extensive library of public domain software in Apple CP/M disk format. This brings to three the number of formats they offer. They can be reached at: CPMUG, The CP/M Users Group, 1651 Third Avenue, New York, NY 10028.

Zero Length File Test

Being able to determine whether a file contains any data before proceeding into your main program logic can be useful. In many batch processing systems it is not at all uncommon for output files to be opened but never written to because of a lack of the appropriate input records. If the next program in the stream were, for example, a print job, it would be more appropriate for it to issue a message stating that no input is available rather than printing headings and totals with no detail records.

I have some experience with this situation occurring with one of the more popular accounting systems. It is difficult to explain a report containing nothing but headings and totals to a customer who confronts you with the question: why print a report when there were no data to be printed?

If you have this need, you may want to insert a short routine (see Listing One, page 100) in your program at the appropriate spot or set it up as a subroutine to be called from BASIC or COBOL.

PAUSEIF, QUITIF, and now SKIPIF

Over the past several months two programs to provide control over submit file processing (PAUSEIF and QUITIF) have been printed in *DDJ*. John Ramsey of Louisville, Kentucky has sent in a third which allows conditional command skipping to be implemented. The use of these three programs together will undoubtedly provide most of the control you may need to ensure that batch procedures are executed properly. A complete listing of John's program can be found in Listing Two (page 101).

Miscellaneous

Several months ago I asked for help in finding patches for the CCP to search user area 0 for .COM files not found in the current user area. Well, I got what I asked for! A number of different routines have arrived which I want to share with anybody that is interested. If you send me a SASE #10 envelope (business size) I will be happy to return to you what I have.

A lot of good stuff is coming in future issues. I am just about ready to rewrite my systems BIOS for CP/M Plus which I will be putting into the public domain through this column and in paperware form. After it's well tested, I will send it to either CPMUG or SIG/M where you can get it on disk. Chat with you next month.

CP/M Exchange (Text begins on page 98)

Listing One

```
; Short routine to test for zero length files
;
fcb       equ       Ø          ;first byte of fcb
fcbdr     equ       fcb        ;drive byte of fcb
fcbfnam   equ       fcbdr+1    ;first byte of file name
fcbfext   equ       fcbfnam+8  ;first byte of file name extension
fcbex     equ       fcbfext+3  ;current extent number
fcbs1     equ       fcbex+1    ;internal use byte
fcbs2     equ       fcbs1+1    ;status byte used by open
fcbrc     equ       fcbs2+1    ;record count for this extent
fcbdm     equ       fcbrc+1    ;first byte of data map
fcbcr     equ       fcbdm+16   ;current record number
fcbrØ     equ       fcbcr+1    ;16 bit random record number
fcbr2     equ       fcbrØ+2    ;overflow from RØ
;
;
open      equ       15         ;BDOS open function
bdos      equ       Ø5h        ;BDOS jump point
;
;
          ld        de,pfcb    ;load DE register with address
          push      de         ;of initialized fcb and save

          ld        c,open     ;BDOS open function
          call      bdos       ;open file
          stc                  ;set carry flag to indicate error
                               ;if file was not found
          inc       a          ;was file available
          ret       z          ;no - test for carry at return
                               ;point to determine whether file
                               ;was available
          pop       ix         ;load IX register with fcb address
          ld        a,(ix+fcbrc)   ;load a register with number
                                   ;of records in first extent
          or        a          ;set condition codes for testing
                               ;at return point
                               ;Z = zero length file
                               ;NZ = non-zero length file
          ret                  ;back to caller
```

End Listing One

Listing Two

```
;
; SKIPIF -- a submit utility for conditional command skipping
;
;  written by John C. Ramsey
;
;  major parts shamelessly stolen from:
;    QUITIF in Dr. Dobb's Journal, No. 71, Sept. 1982
;    written by Don Wright of San Antonio, TX
;
; Usage
; -----
;  SKIPIF AMBIG $1       -- skip next line on ambiguous name
;  SKIPIF EXISTS $1      -- skip next line on pre-existing file
;  SKIPIF MISSING $1     -- skip next line on non-existent file
;  SKIPIF NULL $1        -- skip next line on null parameter
```

```
;  SKIPIF ZERO $1        -- skip next line on zero length file or missing
;
; Only the first letter of the keyword is checked.  A letter
; other than A, E, M, N, or Z (or an omitted keyword) will print
; the help message and have no effect on the $$$.SUB file
;
TPA     EQU     0100H   ; beginning of program area
BDOS    EQU     0005H   ; bdos entry point
FCB1    EQU     005CH   ; the default FCB in low storage
NAME1   EQU     005DH   ; filename number 1 (keyword)
FCB2    EQU     006CH   ; second parameter, treated as FCB
NAME2   EQU     006DH   ; filename number 2 (argument)
DMA     EQU     0080H   ; buff for read/write
;
PRTSTR  EQU     9       ; print string until $
SELECT  EQU     14      ; select disk function
OPEN    EQU     15      ; open a file
CLOSE   EQU     16      ; close a file
SEARCH1 EQU     17      ; search for first matching name
ERASE   EQU     19      ; delete file
READ    EQU     20      ; read sequential
WRITE   EQU     21      ; write sequential
MAKE    EQU     22      ; make a file
RENAME  EQU     23      ; rename a file
SETDMA  EQU     26      ; set DMA address
;
CR      EQU     13      ; carriage return
LF      EQU     10      ; line feed
EOT     EQU     '$'     ; end of string marker
DISKA   EQU     0       ; function parameter for drive A:
;
        ORG     TPA
;
; Look for files on disk A:
;
DISK:   MVI     E,DISKA
        MVI     C,SELECT
        CALL    BDOS
;
; Check for presence of  $$$.SUB on disk A:
;
        LXI     D,SUBFILE       ; Set FCB area
        MVI     C,OPEN          ; Open file
        CALL    BDOS
        INR     A               ; Is there a file?
        JNZ     XIF             ; Yes, so continue program
;
; Explain the use of the program to user
;
EXIT:   MVI     C,PRTSTR        ; Load help message
        LXI     D,HELPMSG
        CALL    BDOS            ; ...and tell user about SKIPIF program
        RET                     ; exit to CCP
;
;
XIF:    LXI     H,NAME1
        MOV     A,M     ; get function code letter
        LXI     H,NAME2 ; point to filename/parameter
```

(Continued on next page)

Listing Two

```
;
        CPI     'A'     ; SKIPIF  AMBIGuous name
        JZ      AMBIG
        CPI     'E'     ; SKIPIF  EXISTS name
        JZ      EXISTS
        CPI     'M'     ; SKIPIF  MISSING name
        JZ      MISSING
        CPI     'N'     ; SKIPIF  NULL name
        JZ      NULL
        CPI     'Z'     ; SKIPIF  ZERO length file
        JZ      ZERO
;
; Keyword is none of the above, explain use of the program on way out
;
        JMP     EXIT
;
;
; skip next line if the second parameter is an ambiguous name
;
AMBIG   MVI     A,'?'
        CMP     M
        JZ      SKIP    ; it is one -- go do skip
        INR     L
        MOV     A,L
        CPI     NAME2+11
        JC      AMBIG   ; continue to end of filename
        RET             ; unambiguous -- return to CCP
;
; skip next line if the named file does not exsist
;
;
EXISTS  CALL    SEARCH  ; look for the file
        CPI     OFFH    ; was there one?
        RZ              ;   no -- return to CCP
        JMP     SKIP    ;   yes -- skip the next line in submit file
;
; skip next line if the named file does not exist

;
MISSING CALL    SEARCH  ; look for the file
        CPI     OFFH    ; was there one?
        RNZ             ;  yes -- return to CCP
        JMP     SKIP    ;  no -- skip the next line in submit file
;
; skip next line if the second parameter was null, i.e. all blank
;
NULL    MVI     A,' '
        CMP     M       ; test filename
        RNZ             ; ... not null, return to CCP
        LXI     H,NAME2+8
        CMP     M       ; test file extension
        RNZ             ; ... not null
        LDA     FCB2    ; test for a drivecode alone
        ORA     A
        RNZ             ; ..."d:" given alone
        JMP     SKIP    ; drive, name, extension, all null
;
; abort if the filename has a length of Zero
;
```

```
ZERO    CALL    MISSING ; First of all, is file there?
;
        MVI     C,OPEN  ; Open it if so
        LXI     D,FCB2
        CALL    BDOS
;
        MVI     C,READ
        LXI     D,FCB2
        CALL    BDOS
        ORA     A
        RZ              ; read was successful, return to CCP
;
        JMP     SKIP    ; skip next line if end of file occurred
;
; Delete next record by recopying file $$$.SUB to $$$.TMP
;    except for the last record (which is the next command)
;       then delete old $$$.SUB and rename $$$.TMP to $$$.SUB
;
SKIP:   MVI     C,ERASE
        LXI     D,TMPFILE       ; Make sure $$$.TMP isn't there
        CALL    BDOS            ; before making it
;
        MVI     C,MAKE
        LXI     D,TMPFILE       ; Make $$$.TMP (thus opening it)
        CALL    BDOS
;
        MVI     C,SETDMA        ; This is the default but do anyhow
        LXI     D,DMA           ; point to 80h buffer
        CALL    BDOS
        MVI     C,READ          ; read first record in file (last command)
        LXI     D,SUBFILE
        CALL    BDOS            ; info now at DMA at 80h buffer (#1)
;
        MVI     C,SETDMA

        LXI     D,LASTR         ; location of buffer #2
        CALL    BDOS            ; change DMA address to buffer #2
        MVI     C,READ
        LXI     D,SUBFILE       ; read record #2 into buffer #2
        CALL    BDOS
;
        ORA     A               ; end of file?
        JNZ     CLEAR           ; if so go finish up
;
; staggered transfer -- by switching the DMA between 2 record buffers
;    this part transfers $$$.SUB to $$$.TMP BUT with an offset of one,
;    (read #3, write #2). When EOF of $$$.SUB is reached, we exit the
;    loop without WRITing the last record, (which is the next command)
;    thus SKIPPING it.
;
TRANS:  MVI     C,SETDMA        ; work with buffer #1
        LXI     D,DMA
        CALL    BDOS
        MVI     C,WRITE         ; get rec from #1, write to TMP
        LXI     D,TMPFILE
        CALL    BDOS
        MVI     C,READ          ; read from SUB, put rec in #1
        LXI     D,SUBFILE
        CALL    BDOS
```

(Continued on next page)

CP/M Exchange (Listing continued, text begins on page 98)

Listing Two

```
;
        ORA     A               ; end of file?
        JNZ     DONE            ; if so go finish up without writing last rec
;
        MVI     C,SETDMA
        LXI     D,LASTR         ; work with buffer #2
        CALL    BDOS
        MVI     C,WRITE         ; get rec from #2, write to TMP
        LXI     D,TMPFILE
        CALL    BDOS
        MVI     C,READ          ; read from SUB, put rec in #2
        LXI     D,SUBFILE
        CALL    BDOS
;
        ORA     A               ; end of file?
        JNZ     CLEAR           ;if so go finish up without writing last rec
;
        JMP     TRANS           ; do it again until eof
;
;
;
CLEAR:  MVI     C,SETDMA        ; Clear default DMA
        LXI     D,DMA
        CALL    BDOS
        MVI     C,WRITE
        LXI     D,SUBFILE
        CALL    BDOS
;
DONE:   MVI     C,CLOSE         ; save new $$$.TMP file
        LXI     D,TMPFILE
        CALL    BDOS
;
        MVI     C,ERASE         ; erase old $$$.SUB file
        LXI     D,SUBFILE
        CALL    BDOS
;
        MVI     C,RENAME        ; rename $$$.TMP to $$$.SUB
        LXI     D,RENFILE
        CALL    BDOS
;
        MVI     C,PRTSTR        ; tell user program worked
        LXI     D,SKIPMSG
        CALL    BDOS
        RET                     ; exit to CCP
;
; subroutine to use the search-for-the-first-match function
;
SEARCH  MVI     C,SEARCH1
        LXI     D,FCB2
        CALL    BDOS    ; find first matching name
        RET             ; return A=ff, 00, 01, 02, 03
;
;
SUBFILE DB      1,'$$$      SUB'
        DB      0,0,0,0,0,0,0,0,0,0,0
        DB      0,0,0,0,0,0,0,0,0,0,0
        DB      0,0,0,0,0,0,0,0,0,0,0
```

```
;
TMPFILE DB      1,'$$$     TMP'
        DB      0,0,0,0,0,0,0,0,0,0,0
        DB      0,0,0,0,0,0,0,0,0,0,0
        DB      0,0,0,0,0,0,0,0,0,0,0
;
RENFILE DB      1,'$$$     TMP'
        DB      0,0,0,0
        DB      1,'$$$     SUB'
        DB      0,0,0,0
;
SKIPMSG DB      CR,LF,'The above condition is true.',CR,LF,CR,LF
        DB      'The next line has been DELETED.',CR,LF,EOT
;
HELPMSG DB      '^[*'
        DB      'SKIPIF is a program to be used with SUBMIT',CR,LF
        DB      CR,LF
        DB      'Usage:',CR,LF
        DB      '------ ',CR,LF
        DB      '  SKIPIF AMBIG file   - skip next line if ambiguous name'
        DB      CR,LF
        DB      '  SKIPIF EXISTS file  - skip next line if pre-existing file'
        DB      CR,LF
        DB      '  SKIPIF MISSING file - skip next line if non-existent file'
        DB      CR,LF
        DB      '  SKIPIF NULL file    - skip next line if null parameter'
        DB      CR,LF
        DB      '  SKIPIF ZERO file    - skip next line if zero length file'
        DB      CR,LF,CR,LF
        DB      'Only the first letter of the keyword is checked.  A letter'
        DB      CR,LF
        DB      'other than A, E, M, N, or Z (or an omitted keyword) will'
        DB      CR,LF
        DB      'display these instructions but have no effect on the submit'
        DB      CR,LF
        DB      'file.  Parameters or actual file names may be used in'
        DB      CR,LF
        DB      'the submit file as arguments to SKIPIF.'
        DB      CR,LF
        DB      EOT
;
; data area
;
LASTR:  DS      128     ; buffer
;
        END     XIF
```

End Listing Two

ACTxx Cross Assemblers

From SORCIM, 2310 Lundy Avenue, San Jose, CA 95131
Requires CP/M 1.4 or better with 32K of RAM
$175 each
Reviewed by Steve Newberry

For some time now I have urged that microprocessor manufacturers should make 8080-hosted cross-assemblers for their processors available if they are really interested in selling *chips* (instead of development systems). For every development system capable of producing code for a given processor, XYZ, there are at least ten 8080/Z80-based systems running on CP/M, and I leave it to you to guess at the number of 8080/Z80 products that have been developed on these systems.

Alas, to no avail. Cross-assemblers that have been offered in the past have generally been written in ANSI Fortran and only been accessible on the maxicomputers – at nontrivial hourly rates. The motives for such a policy may have seemed plausible at the time, but the practical consequences have proven to be less than ideal. In order to do development work for the XYZ processor one had to have either access to an XYZ development system running the resident assembler or to a mainframe (usually a 370) running the cross-assembler. Either approach imposed an initial overhead (for rental fees alone!) of such magnitude that hundreds (thousands?) of projects which might have been developed for the XYZ were never undertaken. One may certainly speculate about the consequent loss of sales (and jobs!) which might have resulted.

Fortunately, the small software houses are beginning to address themselves to this need, and it is now possible to purchase 8080-hosted cross-assemblers for all the major micros except the Z8000 and the MC68000.

Sorcim, the producer of Pascal/M and Supercalc, has for nearly two years been distributing a family of cross-assemblers which run on the 8080 under CP/M and generate code for 8080, Z80, 6502, 6800, 8086/8088, and 6809. The family is called ACT ("Assembly Coded Translator"), and each assembler package is called ACTxx (where "xx" is one of: 80, 65, 68, 86, or 69). The current release, Version 3.5, consists of a CP/M-compatible, 8-inch diskette and a 70-plus page *User's Reference Manual.* The ACTxx diskette contains seven files (eight in the case of ACT80):

- ACTxx.COM – assembler
- DOCOPxx.asm – source file containing all ACTxx mnemonics
- IPARAMS.acd – file of definitions common to all assemblers
- ACD/EG.asm – example demonstrating use of LINK pseudo-op
- UTIL.acd – utility subroutines (ACT-80 only)
- DATTIM.com – sets Date and Time for ACTxx listing header
- INSTALLA.sub – copies ACT system to system disk under CP/M
- INSTALLC.cmd – copies ACT system to system disk under CDOS

The manual is punched for a three-hole binder. There is no index, but the Table of Contents is intelligently done, and all the information is conveniently accessible. The command line to CP/M invoking ACTxx follows the same protocol as that of Pascal/M, for example:

A>ACT80 <filename> <cr>

assembles <filename>, puts the (absolute) hex file on the selected disk under the name <filename>.hex, and doesn't give you a listing unless you ask for it.

Assembly is extremely rapid. I used RELOC.ASM (from the CPMUG library) as a benchmark and timed ACT80 against the CP/M assembler ASM. (ASM is *really* fast.) RELOC is a 10K file which ASM assembled in 16 seconds as compared to 18 seconds for ACT80. (The tests were run on a 4 MHz Z80 with a double-density 8-inch floppy disk.) That means that the very substantially increased power of ACT80 over ASM (macros, many pseudos) is purchased at about a 10% increase in assembly time. (MAC also ran the same job in 18 seconds.)

Although the other cross-assemblers were not compared for speed, they should be comparable because of the way in which they are implemented. The whole macro expansion portion is constant from one assembler to another; the only thing that changes is the table containing the pseudo instructions, the target machine mnemonics, and the corresponding machine codes.

8080-Resident and Z80 Cross-Assembler

ACT80 will assemble both for the 8080/8085 and for the Z80. Originally, the assembler had been produced for Sorcim in-house use in writing Pascal/M. Since the programmers were already familiar with the mnemonics for CDC mainframes, the Sorcim mnemonics initially had a distinctively CDC-like flavor. When it was decided to market the ACT family of cross-assemblers, the mnemonics were extended to include the entire Intel set (including the '85 RIM and SIM instruction). The Zilog set is still only partially represented, although there are "Zilog-like" mnemonics which complete the Z80 instruction set. However, the Sorcim mnemonics regard movement between registers as moves (MOV), but memory-register moves as loads (LD), register-memory moves as stores (STO), and constant-register moves as load-constants (LK).

In other words, Z80 programs written on ACT80 can be assembled with only a little massaging (a few macros in ED or WM), by translators which recognize the standard Zilog set of mnemonics, but transforming source programs in the Zilog notation to the Sorcim set is rather tedious.

6502 Cross-Assembler

Turning to the other cross-assemblers, we find that ACT65 recognizes the MOS Technology mnemonics with only three exceptions, and these are trivial to translate in either direction:

(1) MOS Technology encloses address expressions in round parentheses where Sorcim uses square brackets.

(2) Some 6502 assemblers use a single prefixed quote-mark (apostrophe) to signal a one-character string. Sorcim requires that all strings be enclosed between balanced quote-pairs.

(3) In the Sorcim assemblers, the symbols "<" and ">" are only used as *binary* operators, and the corresponding unary operators are "low" and "high."

6800 Cross-Assembler

The ACT68 mnemonics are a compatible superset of the Motorola 6800 set, and only minor differences exist in expression evaluation and pseudos:

(1) ACT68 observes normal operator precedence.

(2) ACT68 requires that quote marks demarcating character strings be

paired (as in the ACT65 case noted above).

(3) ACT68 does not support

NAM (use TITLE instead)
MON
OPT (use LIST instead)

(4) ACT68 requires that comments be preceded by semicolon (or asterisk if in column one).

8086/8088 Cross-Assembler

To appreciate the difference between the Intel and the Sorcim 8086/8088 assembler mnemonic sets, it is helpful to know something about the structure of the Intel assembler MCS-86.

In designing a mnemonic instruction set, one usually attempts to strike a balance between two desirable but mutually incompatible ends: (1) a small simple fast assembler program, and (2) a small simple easily learned set of mnemonic operator names.

When the instruction set is small to begin with, the problem is fairly tractable, but as the instruction set grows, it becomes necessary to resort to the device of "operator overloading" to keep the distinct mnemonics down to a reasonable number. This means that the assembler must be capable of recognizing that an operator can take several different categories of operand and be able to distinguish these categories correctly.

For example, the Zilog mnemonics for the Z80 compress all the LOAD, LOADCONSTANT, STORE, and MOVE operations into the single mnemonic "LD," and give to the assembler the responsibility of distinguishing between registers and memory and constants, as well as operand lengths (byte versus word). This makes for a larger, more complex (expensive), slower assembler, but also makes it much easier for the programmer to learn the machine.

The designers of the Intel MCS-86 assembler saw fit to "make a virtue of necessity" by making the MCS-86 assembly language a *strongly typed* language. Thus, in MCS-86 (as in Pascal and Ada) the assembler looks up the number of bytes assigned to a variable every time that variable is accessed in the source code, and checks that only arguments of the same length may be acted upon by a single operation. (There is an override, but it is not relevant here.) Aside from enforcing an additional level of assembly-time error checking, this also permits the large instruction set of the 8086 to be compressed into a much smaller, highly overloaded set of mnemonics.

Sorcim adheres to the Intel mnemonics *almost* as nearly as can be accomplished without actually going over to the strongly typed restrictions of MCS-86. This is largely accomplished by distinguishing word from byte operations through a "B suffix" convention. In effect, the programmer learns (almost) the same set of mnemonics, *but* is required to inform the assembler when byte operands are forthcoming by suffixing the letter "B" to the operator mnemonic. Thus, an ADD instruction is applied to byte (rather than word) operands by writing ADDB. The other differences are (1) ACT86, like its siblings, also differentiates between register-register MOVes, register-memory STOres, and memory-register LoaDs; and (2) the "destination, source" ordering of operands is reversed in the case of the STOre instructions.

Here again, the practical consequences are that it is easy to transport ACT86 source code to MCS-86 (by defining macros to delete all the "B suffixes," replace the LD and STO, etc.) but rather tedious to port in the other direction.

6809 Cross-Assembler

The remarks concerning ACT68 apply here as well. There are two additional pseudos: SETDP (sets the assembler's pseudo Direct Page Register), and FAIL (synonym for ERR). ACT69 also does not support the Motorola REG pseudo. Otherwise the Sorcim mnemonics are a compatible superset of the Motorola standard.

Features Common to All Members of the Family

Command line options allow you to: specify the disk drive to which the hexfile will be sent (or you may send no hexfile at all), specify the destination of the listfile (lineprinter, console, diskdrive, or no listfile), set page size (defaults to PS=58), set line length (defaults to SL=140), specify either of two flavors of cross-reference map (default lists globals only), specify library files to be linked in with the program, and specify the origin (ORG) at which the program is to be loaded. The manual is very clear about how to specify these options.

The pseudos include the standard set you'd expect to find: DB, DW, DS, ORG, EQU, SET, PAGE (or EJECT), TITLE, SPACE, the conditional assembly controls IF, ELSE, ENDIF, a toggle (LIST) to control listings (invaluable in debugging macros), END, LINK (approximately equivalent to $INCLUDE or MACLIB), MACRO/ENDMACRO, and several built-in macros. Macros may take up to nine parameters and may be tested.

In addition to these are several pseudos which call for special comment. LOC permits you to assemble a chunk of code to LOAD at a different address than that at which it is to execute, load it at the ORG address, then move it to the LOC address and execute. This sort of thing arises when programs or pieces of programs in ROM need to be moved to RAM for execution, or in writing transient device drivers which are to be relocated to lie just beneath the operating system. You may not use LOC very often, but when you do need it you'll bless the designer of ACT for having provided it.

Another nice touch is VFD. This permits you to write your own assembler (for some other processor) in an *intelligent* manner. Of course, the macro facility would allow you to grind out an assembler by writing a separate macro for every possible opcode – but this would entail writing out hundreds (or even thousands!) of separate macros. VFD permits you to specify arbitrary bit-fields within a byte. In this way you can construct your assembler the way it *should* be done: as a fixed pattern of bits for each generic opcode, with variations in particular bit-fields specifying the source/destination operands. (To find a feature like this in a $175.00 assembler is extraordinary!)

The file IPARAMS.acd is a library file of symbol definitions: EQU's for the standard ASCII control characters and the CP/M system-call functions and magic entry-points. I found it convenient to edit in an "END" statement and adopt the convention that *all* programs are terminated with a "LINK IPARAMS.ACD" statement. The file UTIL.acd (found only in the ACT80 package) contains several flavors of string move/compare, block-fill, and other commonly used functions.

Another thing I like about these assemblers is their ability to treat all letters as though they were upper case without forcing lower-case to upper-case conversion on the listing. It's much easier to read a listing when it looks like the original source! The ACT assemblers also allow you to use "_", ".", and "#" as separation ("break") characters. ACT treats the underline "_" as a "ghost" character (it is not significant). Thus, you can write variable names and labels as things like "iteration_#" or "error.exit". All this makes it convenient to write very *readable* source and to generate very *readable* listings. And that means time and money saved in development and maintenance.

DDJ

Sorcim also provides the same packages for a variety of other chips besides the 8080. Contact them for details. – Ed.

BOOK REVIEWS

Microcomputer Disk Techniques
By Paul Swanson
Published by BYTE/McGraw-Hill
234 pages, $15.00
Reviewed by George W. Jolly

When you bought your shiny new computer, you probably picked one with as much phaser power as you could find. You took it home, connected all the cables, and found you could zap klingons with the best of them. Soon, however, you noticed something, a voice, whispering in your ear. You tried to ignore it, but each day it grew louder and louder. Eventually you were forced to face the question, "How can I do something useful with this thing?"

Stoically, you reached for the BASIC manual that came with the computer and began to read. You learned about variables, subroutines and statement labels. With some fascination you read about sequential disk files, which must be read in the same order they were written. You moved on to random-access files, and learned that you could read or write items in any order you pleased. With such tools you could handle nearly any task — once you chose a task!

With all this new knowledge, you set about programming your first application program. As the program grew, you realized that your tools were more detailed than your imagination. Each feature you added to the program made it harder and harder to remember how everything works. Worse, those disk files were not nearly as fast as you had thought. What went wrong?

What probably went wrong is twofold. Firstly, your program was not planned well enough in advance. Good planning keeps the program readable even after it has become a large program. Secondly, the BASIC system only provides very simple methods of accessing disk files. In order to use files well you need to write routines that are more intelligent about disk files than is BASIC itself.

Paul Swanson, who heads a computer consulting firm, effectively shares his experience in both program design and disk file usage. He presents source listings for subroutines that implement several different file structures. Through the book, these routines are modified to meet other needs. You can learn as much about program structure in BASIC as about disk handling from this book.

Most of the book is based on TRS-80 BASIC. A large appendix revises the listings for the Apple II and discusses the differences. The ideas discussed are valuable in themselves, and a user of a different language altogether would profit from the reading. My own experience says you can never see too many examples of good design. One often-overlooked issue is the design of the user interface. Questions properly presented to the user are easier to answer, and the program should recognize erroneous entries intelligently. Author Swanson has a good discussion of this topic.

The book presents informal discussions, program listings, and detailed explanations of the programs. Amusing drawings by Eddie Germano enhance the reading experience, as does author Swanson's easy prose style. Concepts are developed from the ground up, and the explanations are clear. This book should be a help in designing useful software.

Microcomputer Design and Construction
By Alan Clements
Published by Prentice/Hall International
520 pages, $32.00
Reviewed by George W. Jolly

Alan Clements has packed this book with surprises, all pleasant. Basically a text about microcomputer systems, this book is really a diary of the author's own design experiences. Although his system is based on the Motorola 6800 CPU, the book's value is not limited to 6800 systems. At every turn unusual tidbits of information appear. The author does an excellent job of presenting engineering concepts to non-engineers.

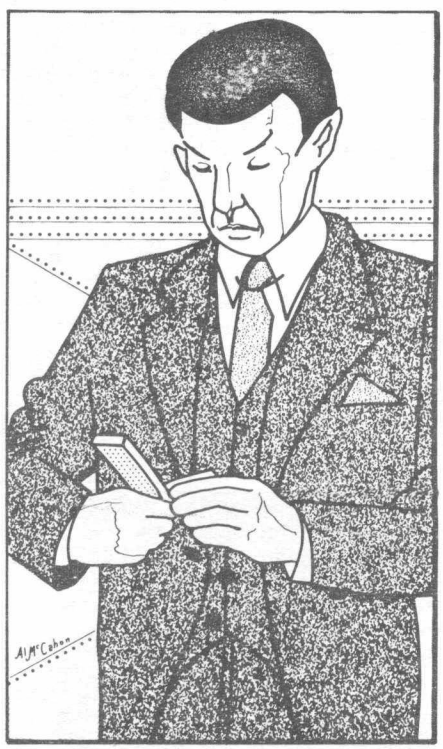
Clements explains the structure of a microcomputer system, showing relationships between the basic components. His explanations of timing diagrams and timing compatibility are excellent. The emphasis is on understanding the subject well enough to design a system yourself. The writing style throughout is clear and the illustrations are most helpful.

One real plus comes from the author's familiarity with computer components. He describes several chips that are not commonly found in magazine articles. One is the field-programmable logic array, a variant of which is used for address decoding in the illustrated system. Another area of interest concerns numeric chips. The author describes the Am9511 numeric processor and the MPY-8HJ 8-bit parallel multiplier. Digital/analog conversion receives very good treatment, and the problem of sampling an audio signal properly is explained well.

One subject which is not as fully developed as this reviewer would have liked is multiple processor systems. The author's example connects a 6802 CPU board to the main system with a serial data link. Although this is an interesting area in itself, the 6800 architecture discussion earlier in the book had hinted at connecting two CPUs directly in the same system. Still, many books unfortunately leave this topic out altogether.

Most microcomputer design books cover the same basic topics, because most microcomputers contain the same basic functions. They also generally omit the same topics, because they are considered too complex for beginning designs.

What distinguishes this book is the depth of information given. For example,



Interfacing to S-100/IEEE-696 Microcomputers, reviewed in an earlier issue, mentioned that the layout of a dynamic memory board is critical, but does not tell why. *Microcomputer Design and Construction* specifically discusses the nature of the problem – dynamic memory chips have rapidly varying power requirements. Their sudden power demands make more noise than the steady, albeit higher, power demands of static memory. Although neither book presents dynamic memory designs, Dr. Clements has room in the textbook format to flesh out the story.

This is a textbook, with exercises at the end of each chapter. It is an excellent book for someone considering designing their first microcomputer system, whether or not he or she uses the 6800 CPU.

Pascal Implementation: The P4 Compiler
By S. Pemberton and M.C. Daniels
Published by Ellis Horwood, Ltd. (Eng.) and by John Wiley & Sons (U.S.)
290 pages, $64.95 hardbound
Reviewed by William G. Hutchison, Jr.

Several years ago I evaluated the cost of implementing P4 Pascal on a minicomputer which shall remain nameless. At that time, there were two serious obstacles: the sketchy documentation supplied with the implementation kit, and the poor quality of the compilers. Today's minicomputer compilers are no better, but the documentation problem is solved.

Steven Pemberton and Martin Daniels have done a thorough job of describing the internal anatomy of the portable Pascal compiler version 4 (P4). They wrote this book with the assistance of Niklaus Wirth himself, originator of Pascal. They are frank about the shortcomings of the compiler. Here is my favorite comment:

> The treatment of the 1 . . 10 case is verging on the abominable. The jump from the *then* part of an *if* statement into its *else* part is probably the worst bit of programming in the compiler. It is unlikely that Pascal even allows it, and it should certainly be rewritten.

This is on page 27, regarding procedure "insymbol." The comment sounds sarcastic, until you realize that it is probably an afterthought from Wirth or Urs Ammann. Actually, this sort of commentary is very valuable. It shows that the book is not superficial and not a whitewash. The authors show flaws and limitations, then point out a way to work around the problem or solve it.

The book is organized one chapter per major section of the compiler. Each chapter begins with theory and description, then gets down to a line-by-line description of the compiler source code. These line numbers refer to the version of P4 which is reprinted in the companion volume *Pascal Implementation: Compiler and Assembler/Interpreter.* You should buy both books to be sure that you can follow the comments. If you get a different release of P4, the line numbers may be different. The publishers of these books offer a tape version of P4. It would make sense to get this for purposes of cross-referencing.

In addition to the commentary, there are helpful diagrams of the data structures used in the compiler and interpreter. The description of the p-code instruction set is a great improvement over the brief brochure included with the source tape.

Obviously, this book is intended for experts. You have no chance of understanding it without a thorough knowledge of Pascal. It probably wouldn't do for a textbook, because parts are pretty terse, and there are no formal exercises. However, an enterprising teacher could build a dandy semester course around it by having the students pursue some of the hints for improving the compiler. These hints are numerous and shrewd. For a year course, how about going whole hog and extending it to ISO standard?

There are a few typos: on p. 30, *if* option.list should be *if* options.list. On p. 37, 1/21975 should be (1975). On p. 48,

b= *record* j:integer; next; ↑a *end* ;

should be

b= *record* j:integer; next: ↑a *end* ;

These typos are not serious and do not alter the sense of the text.

I've saved one of the best features for last. The appendices of the book contain a listing of all procedure and function headers by line number, a cross-reference of uses of error numbers in the compiler, a list of code generation procedure calls by line number, and a symbol cross-reference of the compiler and interpreter (keyed to the line numbers in the other volume). The original authors of P4 should have included these helpful indices with the source, but they didn't. Pemberton and Daniels have come to the rescue.

If you want to implement P4, or learn a great deal about Pascal, buy these books, and save weeks of your time burrowing through listings.

Projects in Machine Intelligence for Your Home Computer
By David L. Heiserman
Published by Tab Books
337 pages, $9.95
Reviewed by Richard L. Lozes

A pleasant surprise. The author was motivated to study machine intelligence by computer simulation after becoming frustrated by the time elapsed in watching a robot explore its environment. The result is a book containing insight, pedagogy, and ready-to-use programs that duplicate his efforts.

The text expands on two genera of "creatures," Alpha and Beta. These are represented by simple graphic characters on the screen. Alpha creatures move at each discrete time step according to a random variable. Beta creatures begin life exactly like Alpha creatures but, by contrast, remember their experiences and draw upon those memories for appropriate responses to previously experienced situations. The reader will recognize these as examples of Markovian and non-Markovian processes, respectively.

There are, further, two species associated with each genus, viz. pacifist and killer. One can quickly imagine the complexity which communities, especially integrated communities, of such creatures (along with various "inanimate" objects) might exhibit in certain phases of their life cycle.

The nicest thing about the book is that the code (in BASIC) is all here, ready to go. It is supplemented by clear flowcharts. The reader can immediately explore the creatures, collecting data along the way both by simple observation and with the aid of included subroutines.

The tools here are definitely worth serious consideration. First, the conceptual framework is exceedingly simple, and fascinating behavior already is evidenced. Second, because of the first point, an infinity of variations await us.

Alpha and Beta creatures only suffer local (nearest neighbor) interactions with the environment. Introduction of various forms of action at a distance would certainly yield different behavior and would likely prompt heated discussion of topology.

Beta creatures remember only the last occurrence of a given situation. Thus they are governed by joint probability functions of order two. Why not three? or more? In this way, a creature's history would more directly influence its present behavior.

Forgetfulness according to a stochastic function of time could simulate effects which we are all familiar with. And a stochastic death function could lead to models of population dynamics, especially when complemented by the introduction of a new species – promiscuous.

Should I say anything negative about *Projects . . .* ?

No.

A Practical Introduction to Computer Graphics
By Ian O. Angell
Published by John Wiley & Sons
146 pages, $16.95
Reviewed by Susan Bowers

A Practical Introduction to Computer Graphics explains in mathematical terms how to create graphical representations of complex concepts. It progresses from two-dimensional graphics and geometry to coordinate geometry, three-dimensional space, and orthographic projections. There is a chapter on techniques for making computer movies, and one containing an interesting list of suggested projects, plus a concluding chapter on what comes next in graphics.

This technical look at computer graphics assumes the reader has a working knowledge of Fortran IV and the mathematical concepts of cartesian coordinate geometry. It is essentially a textbook for advanced graphics students who are also advanced mathematics students. In addition, it is very machine-specific, assuming the use of "the standard Tektronix 4010 scope" and access to the Calcomp library routines.

The book is very well written, with concise explanations between each of the graphics assignments. Assuming that access to the specific equipment and programs is necessary, by following the examples and explanations, a student of advanced mathematics should be able to create intricate graphical representations.

The machine-dependent features would tend to limit the audience of this book. However, the graphics concepts in it might be very useful in architectural design and planning, industrial design, space technology design and futuristic art. In addition, the excellent chapter on techniques for creating computer movies could have application in educational films, TV, and advertising.

Word Processing Buyer's Guide
By Arthur Naiman
Published by BYTE/McGraw-Hill
325 pages, $15.95
Reviewed by D. E. Cortesi

This book attempts to be what its title suggests: a help to the person who is shopping for a word processing system. In the main, it succeeds very well. Naiman sets out to cover four topics. First, of course, come the inevitable chapters on "What is a word processor and why should I want one?" He conveys the joys and advantages of computer-aided writing. Then, aided by many photos, he introduces the component parts of all w-p systems, whether they be dedicated ones or personal computers. The orientation lessons are well done; they should give the confused shopper a good grasp of features, terminology, and functional trade-offs.

The second topic is how to evaluate the software. Naiman devotes a long chapter to his own stringent, detailed evaluation procedure. He supplies blank checklists and instructions on how to use them. The shopper with the time and patience to apply his procedure in full should certainly end up with the right system. Few people will have that much time to spare; even so, just seeing that an objective rating is *possible* should be a help, and Naiman's checklists make a good basis for one's own, shorter version.

Naiman's fourth topic is the selection of hardware. Here again, he describes all possible features at length, and provides detailed checklists for evaluation.

No, I didn't forget the third topic; I saved it for last because it's the most fun. Naiman requested input from 70 vendors of w-p systems (54 of them publishers of personal computer software; the rest makers of dedicated systems). He got responses from sixteen, and managed to gather enough detailed information to apply his evaluation procedure to 14 products. These evaluations, with his comments and justifications on the relative point scores, are the heart of the book.

Any such comparative ranking is bound to be controversial. I looked ahead to see where my own choice of software came in, and found it in 13th place out of 14. Then I double-checked every step of Naiman's procedure, and he won me over.

His tests are comprehensive, and he has applied them rigorously and fairly. Provided that you allocate points to features in the proportions he does, you will come out with the same scores. (Two dedicated systems, CPT and Dictaphone, score highest. The best-scoring personal computer package is Pie Writer, a close third. EasyWriter version 1.0 trails the field with 35 of 100 points.)

Naiman lists 90-odd other products, giving for each its vendor's name and address, and adding short comments for those on which he had any data. He notes *InfoWorld* ratings where available. And he is careful to note every product whose vendor failed to respond to his letters of inquiry. There is a hint of revenge in this, but I understand his frustration very well. As Naiman says, "If a company can't answer a business letter, do you want to depend on them for support?"

In summary, I liked *Word Processing Buyer's Guide* very much. It is firmly on the consumer's side; it delivers solid, practical advice in a clear, informal style; and it tells how (and better, shows how) to rate products objectively. It is just the kind of book that the personal computer industry needs.

OF INTEREST

by Michael Wiesenberg

Z80 and 80186 in Portable Package

The **Nano**™ and **Pico**™ series computers from **SKS Computers** are portables in real leather briefcases. The SKS 2502 Nano™ is compatible with IBM and with SKS's line of desktop computers. It has a Z80A, 80K RAM, a 5-by-9 built-in green screen that displays 16 or 24 lines by 80 characters, two RS-232C serial ports, and two 5¼-inch, 400K floppies. The software includes CP/M, modified C BASIC, Perfect Writer, Menu Runtime, and Filer, Speller, and Calc. The package costs $2495. With dual processors, adding the 16-bit Intel 80186, Nano™ costs $3295, and has 128K RAM. You can add a parallel port and 5¼-inch, 5Mb Winchester. With the 80186, MS-DOS is standard; Oasis and CP/M-86 are options. The SKS 252 Pico™ is even smaller, weighing 22 pounds. It is the same as the Nano™ series, except that the 5¼-inch drives are replaced with two 3½-inch, 200K microfloppies. However, it is not compatible with SKS's desktops. **Reader Service No. 105.**

For Altos, More Memory, More Storage, and XENIX, Too

Altos has expanded its **586** family of multi-user microcomputers, adding a model with 5¼-inch Winchester, 40Mb, optionally expandable to 80Mb. The **586-14** has a 10MHz 8086, runs XENIX™, and, like the present 12Mb (expandable to 24Mb) 586-10, supports both Altos-Net and Ethernet local-area networking. It also has 3270, 3780, X.25, and SNA protocols. Main memory is 512K, expandable to 1Mb RAM. Standard also are an intelligent I/O controller, memory management, six serial ports (expandable to ten), a battery clock and calendar, Multibus-like architecture, power failure detection (but they don't say what happens once it is detected), and that favorite buzzword of makers of business computers, *internal distributed intelligence*. The 586-10 costs $7990 and the 586-14, $10,990. **Reader Service No. 107.**

Turn Your IBM PC Into a Portable

The **Colby PC-1 Conversion Kit** turns the IBM PC into a 26-pound portable, including 9-inch, high-resolution display, switching power supply, wiring harness, interface boards, and a sturdy 15-by-17-by-8½-inch case of molded, flame-retardant Cycolac (the stuff that football helmets are made out of). The conversion can be made by an average user in less than one hour, or the local dealer can do it. After the one-time conversion, the PC chassis can be used as an expansion chassis to hold up to five standard plug-in PC boards, in addition to the five in the Colby unit. The Colby PC-1 costs $899, and an optional battery pack is also available for $489. **Reader Service No. 109.**

Turn Your Portable Into a Stationary Unit

While desktop units are being converted into portables, here comes the **Data-Console** from **Professional Data Systems**, a detachable wooden keyboard enclosure and cpu/monitor cabinet that converts the Kaypro into "a comfortable and efficient workstation in less than two minutes without any tools" (in other words, makes it nonportable). $195. **Reader Service No. 111.**

Winchesters Coming Down

VR Data's Hard Disk III Subsystem is now available for the IBM PC, -XT; LNW-80, -II; Apple II, IIe; Franklin Ace; and TRS-80 I, and III. The disk drive starts at $995 for a 5Mb, formatted, fixed system. Another $150 gets you an adaptor that will include all hardware and software to run the units. Also available are 5Mb removable disk cartridge systems, and 10Mb, 15Mb, 20Mb, and 30Mb configurations. **Reader Service No. 113.**

Printer Ribbons from the Drugstore?

A daisy wheel printer for under $499 is a definite item of interest, but **Star Micronics** feels their standard typewriter ribbon spools are the big selling point. These spools, they claim, change the price of printer ribbons from $20 to around $4, and also eliminate the difficulty of finding replacements. Since the as-yet-unnamed printer uses Underwood-compatible ribbons, the "corner drugstore or all-hours convenience store may have it in stock." (Perhaps I should have italicized "may"; the 7-11 near *my* house doesn't sell typewriter ribbons.) Star Micronics also sells 10- and 15½-inch dot matrix printers. The **Gemini 10** and **15**, they say, "offer better performance than [those] from the 1982 small computer printer sales leader, and at a significantly lower price." (Might they be referring to Epson?) At $399 and $649 respectively, these printers have extra fonts and larger built-in buffers, as does their 80-column thermal printer – the $99 **STX-80**. All the printers come standard with removable tractor feeds and more adjustments than those that (usually) come at extra-cost options. **Reader Service No. 115.**

Talk Back to Your Computer

The newest wrinkle in computer interfacing comes from **SuperSoft**, whose first application for their **VoiceDrive** voice recognition software is available for their ScratchPad financial spreadsheet. For example, instead of typing */d c* at the keyboard, you speak the words, "now delete column." All commands are words (in *any* language – you no longer have to remember which key does what). ScratchPad, with VoiceDrive, sells for $495 – or, bundled with the Tecmar voice recognition board, $995. You need PCDOS and 128K RAM. **Reader Service No. 117.**

Gee Whiz, It's KeyWiz

The **KeyWiz VIP** (Very Intelligent Peripheral) from **Creative Computer Peripherals** is an extra keyboard for Apple, Ace, and TRS-80 Model III computers. With 62 user-programmable keys (31 lower-case and 31 shifted), this keyboard can store up to four 62-key keyboards in its own static memory, making effective 248 progammable key combinations, each holding up to eight characters. Custom keyboards with template overlays for VisiCalc, Multiplan, AppleWriter II, ScreenWriter II, SuperText, Magic Window II, WordStar, Executive Secretary, SVS Word Handler, PIE Writer, Easy Writer, and the Model III's Scripsit are now available. For an extra $40, the company will make you a customized plug-in module. The KeyWiz VIP costs $439. A VisiCalc version, called KeyWiz 83, each of whose keys holds four characters, with numeric keypad, costs $299 (add $40 for word processor functions). The KeyWiz Convertible, which also stores four characters per key, costs $299. It has 31 keys and 248 combinations, and comes with VisiCalc and word processor functions. You get a 10-day trial with a money back guarantee and a full one-year warranty. Add $8 for p&h for any order. **Reader Service No. 121.**

Cheap Bauds

Inmac has a **300-baud modem** for under $130 that only requires a push-button phone and modular RJ-11 jack. The unit has automatic mode selection from originate to answer (and vice versa), an automatic voice or data switch, and an audible carrier detect signal. It is compatible with the Bell 103 and runs on its own 9-volt battery for up to 50 hours of continuous use. Inmac offers a free 100-page catalog of over 2000 computer accessories. **Reader Service No. 123.**

CAW for Columnists?

FirstDraft from **PromptDoc** is computer-assisted writing for CP/M word processing systems. The program asks you what kind of document you wish to create, and then questions you about the kinds of ideas you want to cover, converting your thinking into a structured outline. From the outline it builds a skeletal document which you then flesh out with your word processor. FirstDraft then prompts you through the writing process. When you're done, it automatically generates a table of contents and an index. The DocuMentor option's "boilerplate" patterns can be instantly called up for legal documents, memos, proposals, office procedure manuals, as well as technical and software manuals. FirstDraft costs $195, DocuMentor $295, and both together, $395. I wonder if they have a template for columnists... **Reader Service No. 125.**

Editor Provides Free Accommodations for Jailbird

Actually, that's not what the acronym **EdCompCon** means. It, in fact, is the **Educational Computing Conference** sponsored by the **IEEE Computer Society** Technical Committee on Computers in Education. Uses of computer technology in education, including CAI (computer-aided instruction), teaching and authoring software systems, touch screens, computer graphics, robotics, total systems for education, audio-visual embedded systems, and languages such as Logo, Karel, and Plato, will be subjects of seminars. This all takes place October 18-20 at the Red Lion Inn, which is located, according to IEEE, in "Silicon Valley, California." For those of you who can't find that on your map, the Red Lion Inn is in San Jose. Registration

fees will rise in price after September 15th. The fees are: full conference, $85 ($100 after); student rate, $25 ($40 after); one-day conference, $45 ($55 after); and exhibits only, $10. You can get a buffet lunch for one day for $6; all three days, $15; or attend the "Thursday K-12 Banquet Hosted by Western Region of College Board" for $15. (I put quotation marks around that so you won't think it was *me* who left the *s* off "Board." And, for you non-educators, "K-12" means kindergarten to twelfth grade.) **Reader Service No. 133.**

DEC and TI Become Polyglots

Digital Research now offers these programming languages through their **CP/M Library** for the DEC Rainbow 100 and TI Professional Computer: Pascal/MT+ ($600), PL/I ($750), Level II COBOL ($1600), CIS COBOL ($850), CBASIC ($325), and CBASIC compiler ($600) – all operating under CP/M-86. The CP/M Library is software packages for popular computers which, up till now, have included the IBM PC, DisplayWriter, and those using standard 8-inch IBM 3740 disks. In addition to the languages, the CP/M Library includes: Concurrent CP/M-86 for the PC; CP/M-86 for the PC and DisplayWriter; and the CP/M Card for Apple II. The prices quoted are the same as for those languages on other formats. **Reader Service No. 127.**

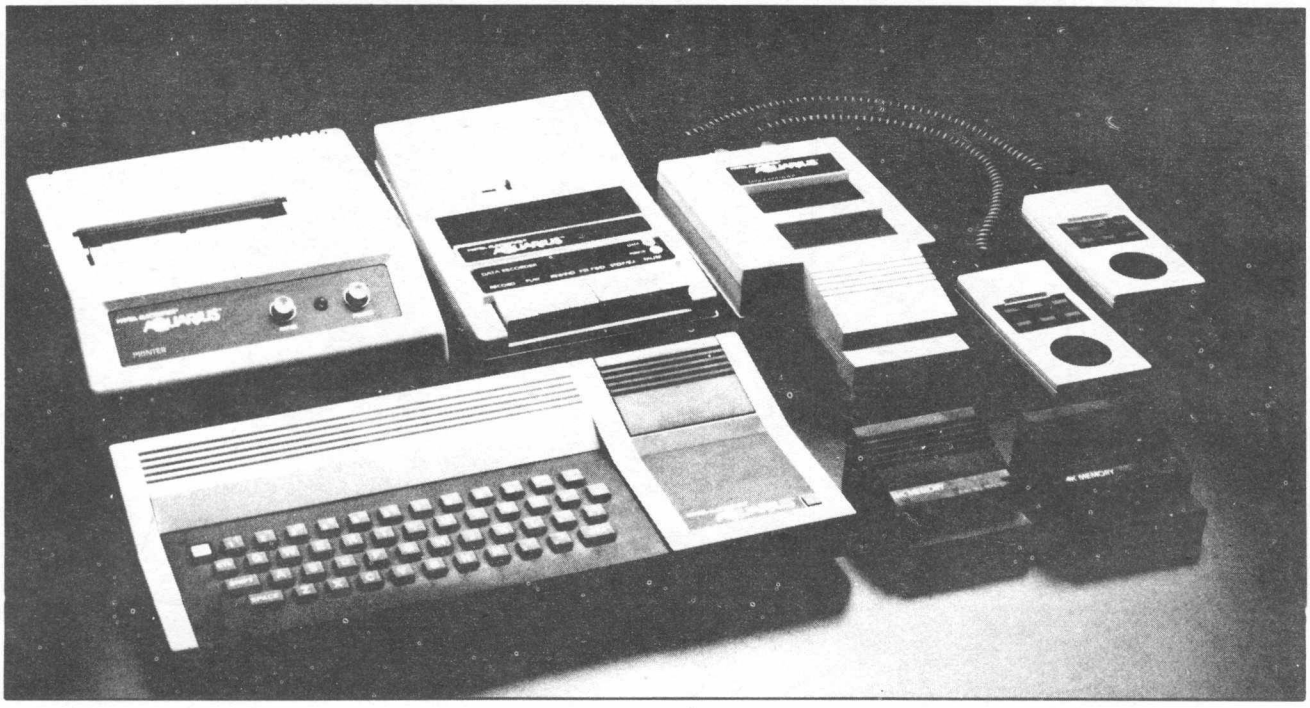
Put a Shell on Your TRS-80

Three of the best features of UNIX are command-line redirection of standard input and output during program execution, pipes (sending the standard output of one program to the standard input of another), and multiple entry of commands on one line. **ZSHELL** from **MISOSYS** does all these for TRS-80 Models I and III with LDSO 5.1, and costs $40 (plus an unspecified charge for shipping). **Reader Service No. 129.**

What Can I Say About This One?

The **CompuGift** terminal look-alikes are a pencil caddy (**Computer Caddy**), **Computer Planter**, and a **Computer Bank**. Each is plastic, 4½-by-6-by-4¼-inch, has an acrylic screen for photo, and comes in beige, dark grey, yellow, and magenta. If you buy in bulk (for which you get a discount), the CompuGift folks will even imprint your company logo on the screen. "The clever gift," they say, "for computer lovers!" (And lovers of computers?) **Reader Service No. 131.**

DDJ



The Age of Aquarius

I spoke last time of the trend toward inexpensive, quality home computers (the under-$200 and under-$100 variety). Now, **Mattel** enters the field with **Aquarius**, a 4K unit that can be expanded to 52K, has 256 graphics characters, displays 16 colors, and has a keyboard with 49 "real" keys. Options include an upper-case and lower-case character capability, graphics thermal printer, data recorder for program storage (sounds like the Vic 20 situation – you must use *their* recorder), a "mini expander" unit for game cartridges, a LOGO language cartridge, and several Intellivision games modified for the Aquarius. It does really seem that this under-$200 computer is aimed for the mass market (and not us tinkerer and programmer types) because nowhere does the press release mention what processor it uses! **Reader Service No. 119.**

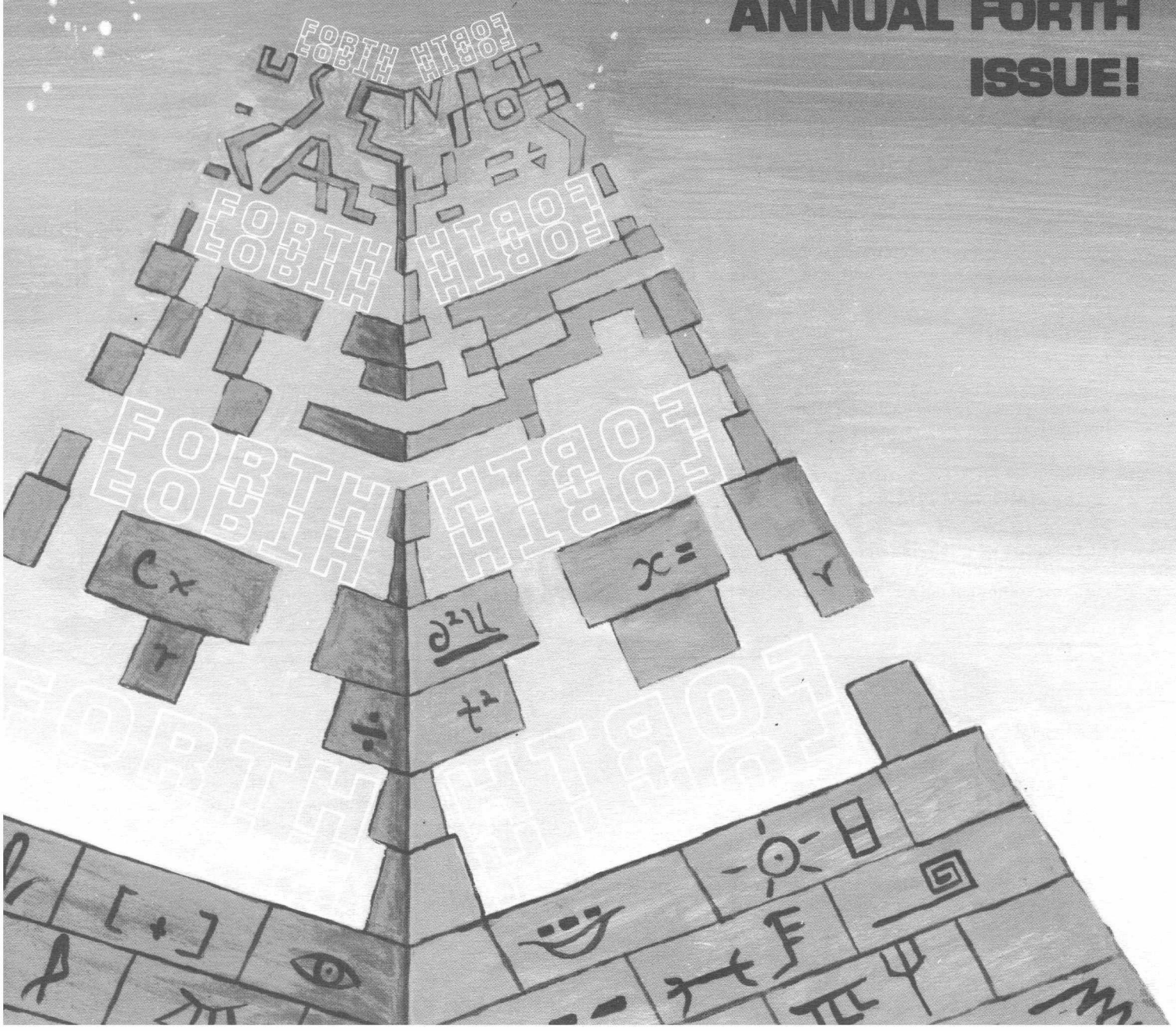
#83 September 1983

$2.95 ($3.50 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems

ANNUAL FORTH ISSUE!



Dr. Dobb's Journal
For Users of Small Computer Systems

September 1983,
Volume 8, Issue 9

18 Forth and the Motorola 68000
by Joe Barnhart
As 68000-based machines proliferate, an increasing number of vendors are offering Forth packages for them. Speaking from experience, this author explains some of the considerations involved in putting Forth on the 68000.

28 A 68000 Forth Assembler
by Michael A. Perry
Any serious Forth system should have a good assembler, and 68000 systems are certainly no exception. This writer discusses the implementation and use of his assembler along with some of the philosophy behind its design.

44 Nondeterministic Control Words in Forth
by L. L. Odette
Programmers usually expect program output to depend on input and the sequence of statements. Making the program counter a random number generator would appear to be disastrous; however, the technique can be both interesting and useful.

54 GO in Forth
by C. H. Ting
While the rules for the game of Go are few, putting it on a computer can be challenging. Author Ting describes his Forth version: how it works and can be improved.

66 SEND and RCV: A Forth Implementation of the XMODEM Protocol
by Robert Taylor III
At the heart of a telecommunications package are the routines for sending and receiving files. The author examines how to implement Christensen's protocol in Forth.

82 Precompiled Forth Modules
by N. Solntseff
Tired of having to wait while the code you are loading from disk compiles? Author Solntseff discusses his scheme for using overlays of precompiled Forth code to speed up the process.

86 Signed Integer Division
by Robert L. Smith
The Forth-83 Standard adopts the "floored" method of integer division. This look at the floored and conventional methods illuminates the reasons for change.

90 When a Page is Not a Page: Forth-83 and Vocabularies
by George William Shaw II
The 83 Standard's introduction of vocabulary search order fell short of defining how to choose the order. The author examines ways and benefits of utilizing multiple vocabulaires in Forth, including an experimental proposal on specification and control of search order.

94 Some Forth Coding Standards
by Edward Wischmeyer
Different programming environments often require variety in the coding methods used. Yet, certain fundamental information must always be conveyed. The author examines some of the factors involved in expressing your ideas clearly and cleanly in Forth.

102 The Forth Sort
by Mark I. Manning
Sooner or later most programmers need to sort data. The author provides Forth implementations for three well-known sorting methods, as well as a program which allows timing comparisons under different data arrangements.

6 Editorial

9 Letters

12 Dr. Dobb's Clinic
How to use a digital spreadsheet program for something fun and unusual: digital circuit simulation.

110 CP/M Exchange
More on disks, drives, and controllers, as well as a way to tell whether CPU-intensive operations are still moving or have hit a snag.

120 16-Bit Software Toolbox
A speed and accuracy benchmark program for high-level languages; a mouse interface for the IBM PC; information on IBM and Epson printer control codes, and more.

128 Of Interest

Dr. Dobb's Journal
For Users of Small Computer Systems

September 1983 Volume 8, Issue 9

Publisher – Jane Laidley
Editor – Reynold Wiggins
Managing Editor – Craig LaGrow
Contributing Editors –
Robert Blum, Dave Caulkins, Dave Cortesi, Ray Duncan, Kim Harris, William Ragsdale, Michael Wiesenberg
Marketing Director – Beatrice Blatteis
Marketing Assistant – Sally Brenton
Advertising Director – Carl Landau
Advertising Sales – Doug Millison
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Billie Greenwood, Lew Griffin, Terri Pond
Production Manager – Barbara Ruzgerian
Production Assistants –
Alice Hinton, Shelley Rae
Typesetter – Paula Fairchild
Cover Illustration – Shelley Rae

Copyright © 1983 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: DeWitt S. Brown, Burks A. Smith, Robert M. Connors, Robert C. Luckey, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Thomas Davis, Jan Steinman, G. Hunter, Ronald E. Johnson, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III, Infoworld, Stan Veit, Western Material Control, S.P. Kennedy, Hatch, Richard Jorgensen, John Boak. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

EDITORIAL

Since our first special Forth issue two years ago, we have seen enthusiasm for the language grow rapidly. An ever increasing number of people are discovering the power and flexibility that it offers. This special issue is our third on the topic, and we are delighted with the amount of support it has generated. As in previous years, we received more good material than we could put into the issue. We would like to thank all those who have contributed their material and expertise.

We have tried to touch upon items that are currently important, as well as some things which should be generally useful or fun. While this issue was being produced, the '83 Standard attracted enough votes for adoption. Two of the articles this month relate to this topic. One examines the change to floored integer division, and one explores an experimental proposal on vocabularies.

Over the past year we have seen the proliferation of computers based on the Motorola MC68000. Vendors have not been far behind in developing Forth systems for these new machines, utilizing the speed and power that this processor offers. We have therefore included two 68000 items – a Forth assembler and a more general piece on implementing a 68000 Forth system.

The rest of the articles should prove both informative and entertaining, such as the ones on precompiled modules, nondeterministic control, and the game of Go. For non-Forth folks, our departments are still in full force this month with their usual interesting information.

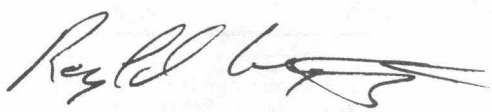
* * *

You may also have noticed that we have suffered our first increase in cover price in over three years. This became necessary as the cost of getting issues to you each month continued to rise. The good news is that only the newsstand price has gone up, NOT the subscription rate. We are hopeful that by this slight increase we can continue to maintain the high proportion of editorial material which we have brought you over the years.

* * *

Looking down the road a bit, we have some interesting items planned. Next month we will be adding a new bimonthly column which will focus on C and UNIX, as well as their derivatives and look-alikes. You will continue to see more on the 68000, programs in C and Small-C, more on graphics, and a few hardware items.

While we have some fine material lined up for you, let me emphasize that we are always interested in new articles and ideas. To obtain our writer's guidelines, send us an SASE. To get our reaction to an article idea, drop us a line. Of course, completed articles are always welcome. And if you have a tidbit, a question, or a discovery, don't forget our columnists. They thrive on reader interaction. It is your contributions that have made DDJ what it is, and it is your contributions that will help us continue to provide the quality material you are looking for.

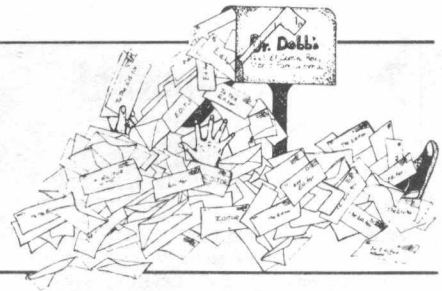


Reynold Wiggins

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to People's Computer Company, Box E, Menlo Park, CA 94025 · 415/323-3111

ISSN 0278-6508

LETTERS



A Better Algorithm for Sieve Benchmarks in Forth

Dear Sir:

As a 68000 Forth vendor, a member of the Forth Standards Team, and as vice president of the Forth Vendors Group, I have been asked repeatedly about Forth performance in the article entitled "Eratosthenes Sieve Benchmarks" (published in *Byte* magazine). I would like an opportunity to discuss the algorithm supplied in the *Byte* article, introduce a better algorithm, provide some timings for our 16- and 32-bit 68000-based products, and make some general comments on the Forth approach to problem solving.

The enclosed Forth code contains three versions of the Eratosthenes Sieve Benchmark as published in *Byte* magazine.

The first version (shown in Figure 1, below) represents the original coding from the article (including a corrected typo). With CSI's 68000 Multi-Forth on an HP Series 200 model 26, 10 passes execute in 22.98 seconds (16-bit version) and 25.77 seconds (32-bit version).

Unfortunately (or fortunately depending on your viewpoint), this algorithm can be better written in Forth. A better algorithm, similar to those optimized for C and Pascal, is shown in Figure 2 (below).

Note that the BEGIN . . . WHILE . . . REPEAT structure in the former has been replaced by a DO . . . +LOOP structure in the latter, taking advantage of Forth's ability to use loop indices as addresses.

This algorithm runs 10 passes in 15.62 seconds (16-bit version) and 17.35 seconds (32-bit version).

In the unlikely event that an application would ever require the rapid computation of the first 1899 prime numbers, the tightest loop would normally be reduced to code using the in-line assembler.

The 68000 code sequence in Figure 3 (below) replaces the internal loop, which traverses the flags array clearing flags for multiples of known prime numbers.

Note that this code procedure replaces six high-level words with eight assembly instructions and utilizes the BEGIN ... UNTIL control structure within the assembler.

This results in the final rendition shown in Figure 4 (below), which executes in 8.86 seconds (16-bit version) and 9.81 seconds (32-bit version).

Industry experience suggests that 10% of the code in a typical application executes 90% of the time (our experience in real time systems suggests that this is closer to 5% and 95%). The current trend in other languages is toward large multi-pass optimizing compilers which optimize the entire program in an attempt to catch the 10% of the code that is in fact worth the effort.

Forth, however, encourages a different approach. All code is typically written and debugged in high level. Then, if necessary, time critical words are recoded with the in-line assembler.

Forth naturally factors applications into a hierarchical set of simple functions (typically under 7 operators or operations per function). These functions pass operands via the stack. Recoding specific words with the in-line assembler, as we have seen in the examples, is a very straightforward process.

The 10% faster execution time of the 16-bit product is typical for most applications, and due to the fewer memory fetches required to handle 16-bit versus 32-bit stack items. Although the 16-bit version complies with the 79 Standard, 10% performance is a small price to pay in consideration of the facts that: the

```
( SIEVE )
8192 CONSTANT SIZE
VARIABLE FLAGS  SIZE ALLOT

: DO-PRIME
  FLAGS SIZE 1 FILL  ( SET ARRAY )
  0 ( 0 COUNT ) SIZE 0
  DO FLAGS  I  +  C@
    IF  I DUP + 3 + DUP I +
     BEGIN DUP SIZE <
      WHILE  0  OVER FLAGS  + C!  OVER  +  REPEAT
     DROP DROP  1+
    THEN
  LOOP
    .  ." PRIMES"  ;
: 10-TIMES CR  ?RTC  10 0 DO DO-PRIME LOOP  .MS ;
```

Figure 1.

```
( SIEVE BENCHMARK IMPROVED ALGORYTHM )
: DO-PRIME.HI  ( -- FASTER ALGORITHM IN HIGH LEVEL  )
  FLAGS  SIZE 01 FILL
  0  ( PRIME COUNTER )   SIZE 0 ( RANGE )
  DO I FLAGS + C@  ( PRIME ? )
    IF 3 I + I + DUP I + SIZE <
       IF  SIZE FLAGS + OVER I +  FLAGS +
           DO 0 I C! DUP +LOOP
                    (FLICK DOWN MODULO I FLAGS)
       THEN  DROP   1+ ( BUMP PRIME COUNTER )
    THEN
  LOOP  .  ." PRIMES" ;
: 10-TIMES.HI CR ?RTC  10 0 DO DO-PRIME.HI LOOP .MS ;
```

Figure 2.

```
 ( CODE ROUTINE FOR SIEVE )
CODE FLICK'EM  ( PRIME#\LIMIT\START -- PRIME#  )
               (   REPLACES:   DO 0 I C! DUP +LOOP )

 SP )+ A1 LONG MOVE,  SP )+  D1 LONG MOVE,
 SP () D0 LONG MOVE,  0 D2 MOVEQ,

  BEGIN, D2 A1 () BYTE MOVE,  DO A1 LONG ADDA,
         A1 D1 LONG CMP,    LT UNTIL,

  NEXT END-CODE
```

Figure 3.

```
( SIEVE BENCHMARK  --  TYPICAL SOLUTION  )

: DO-PRIME.LO
        (  -- FASTER ALGORITHM WITH CODE PROCEDURE)
  FLAGS  SIZE 01 FILL
  0  ( PRIME COUNTER )   SIZE 0  ( RANGE )

  DO I FLAGS + C@
     IF 3 I + I +   DUP I + SIZE <
          IF SIZE FLAGS + OVER I +  FLAGS +
            FLICK'EM  (FLICK DOWN MODULO I FLAGS )
          THEN DROP   1+  (BUMP PRIME COUNTER)
     THEN
  LOOP  .  ." PRIMES" ;

: 10-TIMES.LO CR ?RTC 10 0 DO D0-PRIME.LO LOOP  .MS ;
```

Figure 4.

32-bit system allows unlimited program size; most delivered 68000 hardware is shipped with more than 64K available; and the phone company is now selling 256K RAM parts.

In summary, a better Sieve implementation in Forth provides a 33% speed improvement, while coding the tightest loop in assembly contributes another fifty to sixty percent. One cannot fault the authors of the article; they openly acknowledge far greater proficiency in C and Pascal. It does illustrate, however, the significant impact of programmer proficiency in the language under test on timing comparisons in benchmarks.

I appreciate the opportunity to express these views to your readership, and, as a member of the Forth community, stand up and applaud *DDJ's* continuing involvement in Forth.

Sincerely,
Don Colburn, President
Creative Solutions, Inc.
Vice President
Forth Vendors Group

Forth Debugging from a Full-Screen Editor

Dear *DDJ*,

One of the problems with most debugging aids is that you must continually look back and forth between a listing of the program and the screen to keep track of where you are. This program, however, will add single-step capability to the Laxen full-screen editor (*Dr. Dobb's Journal*, Sept. 1981, page 27) so that the cursor pops from word to word as the program is executed one step at a time. The parameter stack contents are dumped to the bottom of the screen with each program step.

If a bug is found while stepping through the program, you can make immediate changes since you are still in the editor with the cursor positioned at the offending word. After making a change you can immediately resume debugging.

The new function is installed by simply loading screen 98 (Figure 1, below) just before screen 80 in the Laxen editor. Screen 80 contains the CASE: statement, which assigns control key functions. To assign the STEP function to control-W, simply replace the BEEP just after INSERT-MODE in that CASE: statement with STEP.

Pressing control-W will now cause the word (or number at the cursor) to be interpreted as though it had been entered at the keyboard and followed by a carriage return. The parameter stack will be dumped to the bottom of the screen, and the cursor will move right to the next word.

Comments and compile-only words can be skipped over by using the normal cursor control keys. Normally, input parameters should be put on the stack just before calling the editor.

The '79 Standard, MVP Forth system was used with the Laxen editor, but the same technique can be applied to any full-screen editor.

Sincerely,
Tom Blakeslee
Orion Instruments
172 Otis Avenue
Woodside, CA 94062

In Defense of CP/M Plus

In his June Clinic, Dave Cortesi noted some weaknesses in CP/M Plus's sector buffering scheme. Last month, the Intern's direct-access I/O benchmarks produced a much more favorable picture. Below is a response to the June Clinic on behalf of Digital Research which, unfortunately, was received too late to be included in the August issue. While the discussion will have some overlap with the August Clinic, it is still enlightening. – Ed.

Dear Sir:

Digital Research invited me to respond to the CP/M Plus review that appeared in the June Dr. Dobb's Clinic column written by David E. Cortesi. I was not involved with the design or implementation of CP/M Plus, so DRI felt my reaction to the article would be objective.

The column suggests that DRI overlooked several obvious ways to speed up disk accesses, and may have misrepresented or incorrectly documented their product. I was quite impressed by the detective work Mr. Cortesi performed to learn what goes on inside CP/M Plus, but I think his conclusions are incorrect. Let me respond first to some specific claims made in the column, and then address the general question of operating system optimization.

The column reported some trouble in using the MOVE and XMOVE functions to examine internal operating system data structures. Had Mr. Cortesi's SHOWBCB program "played by the rules" of CP/M Plus, however, it would not have been swapping memory banks. CP/M application programs, by definition, execute only within the context of Bank 1. CP/M Plus does leave the overly clever programmer many opportunities to screw up, as did CP/M 2.2. Remember, though, that the BIOS exists to serve the BDOS, so direct I/O calls can (and sometimes do) interfere with higher-level OS functions.

The column "discovers" some things that had already been spelled out in Digital Research documentation and articles, like the fact that CP/M Plus does not actually perform least-recently used (LRU) sector buffering during sequential file I/O. A DRI article that introduced CP/M Plus in the May 1983 issue of *Computer Design* explains why: "However, the operating system overrides the LRU scheme if a series of sequential sectors are referenced. This avoids wiping clean the entire buffer area." The number of buffers consumed for each of the experiments described is consistent with this fact.

The column states that in developing software, "the file one program writes, another program is going to read." A single source file, in turn, may be edited, assembled, linked, and listed. In nontrivial programs, however, changes often affect several source modules, and each module may need re-editing as typing errors are detected. Some *translators* produce a number of intermediate and output files. In practice it's not at all clear which files should be retained by the OS for ensuing programs to use.

Also, software development is not actually a typical CP/M application. Mr. Cortesi and I (and probably most of your readers) do spend much of our time writing new programs, but most CP/M machines are used primarily to run spelling checkers, data base systems, and

(Continued on page 132)

```
SCR #98
  0 ( EDITOR DEBUG, STEP)
  1 : STEP ( --) ( LOADS ONE WORD & DISPLAYS STACK)
  2       BUFPOS DUP    ( START ADR OF WORD)
  3       R-WORD     ( MOVE CURSOR RIGHT 1 WORD)
  4       BUFPOS SWAP -  50 MIN  1-   ( LENGTH)
  5       DUP TIB @ +   0 SWAP !   ( NULLS TO END STREAM)
  6       TIB @ SWAP BMOVE    ( MOVE WORD TO TIB)
  7       0 18 CRTXY     ( MESSAGES ON LOWER SCREEN)
  8       BLK @ >R     0 BLK !   ( SAVE & SEL TERM INPUT)
  9       >IN @ >R     0 >IN !
  A       INTERPRET
  B       40 SPACES   ( CLEAR LINE FOR STACK DISPLAY)
  C       0 17 CRTXY  .S   ( SHOW STACK)
  D       R> >IN !       R> BLK !   ( RESTORE)
  E       0 MOVE-CURSOR   ( RESTORE CURSOR POSITION) ;
  F
```

Figure 1.

DR. DOBB'S CLINIC

by D. E. Cortesi, Resident Intern

The Digital Spreadsheet

Despite all the software available these days, it's still the spreadsheet calculator that most exemplifies the helpful, responsive, intimate nature of a personal computer. Once you've owned a Visi-, Perfect-, Multi-, or Super-Calc, you never want to be without it again.

What's more, new uses for the damn things keep turning up. We've been playing around with one that was new to us, and might be to you as well.

We picked this idea out of the letters column of a recent issue of *Byte*. A previous issue had carried a rather nice digital-logic simulator program written in BASIC. The letter-writer casually pointed out, with no details, that you don't need any special-purpose software; it is just as easy to simulate digital logic with a spreadsheet calculator.

You don't? It is? Hmmm. Wonder what he means? So we started playing around with our well-worn copy of SuperCalc[tm], and darned if he wasn't right. What's more, it's great fun, and might even have a practical use.

Simple Gates

The obvious first thing to try was the simulation of the five simple gates: the And, Nand, Or, Nor, and Xor circuits. And they turn out to be easy, at least with SuperCalc version 1.4 (the exact product and version may be important to this; we have no idea if other spreadsheets will support the expressions shown here).

Provided that you limit all values to being either 0 or 1, an And gate is simply a cell containing

*input1 * input2*

where *input1* and *input2* are the names of other cells. (We will use this convention throughout; an italicized name represents a reference to a spreadsheet cell.) If either input cell contains 0, the value of the And-cell is 0; if both contain 1, its value is 1.

An Or gate is trickier. The simple *expression*

input1 + input2

works, in that its value is nonzero if either input is nonzero, but when both inputs are nonzero, its value exceeds 1. Fortunately, SuperCalc follows BASIC in that the value of a relational expression is a number, 0 for false and 1 for true. So an Or gate becomes

0 < (*input1 + input2*)

which has the value 0 only when both inputs are 0, and the value 1 otherwise. We recast our standard And to the same format, just in case some floating-point near-zero should creep into a simulation:

0 < (*input1 * input2*)

The negated forms Nand and Nor fall out of this very nicely, just by changing the relational function from less-than to equals. Here are all four gates:

And: 0 < (*input1 * input2*)
Nand: 0 = (*input1 * input2*)
Or: 0 < (*input1 + input2*)
Nor: 0 = (*input1 + input2*)

They are easily extended to more inputs. An Xor gate is simply the not-equals relationship:

Xor: *input1 <> input2*

One beauty of a spreadsheet is the ease of building a truth-table for one-cell functions like these. Assign a row to each input, with an appropriate label in column A, and enter all combinations of constant values:

Input A: 0 0 1 1
Input B: 0 1 0 1

Enter the function expression in the third row, supplying the names of the cells above it for its input values. Then replicate the expression, with automatic adjustment of references, across the row. Voilá. a truth table (and instant insight for a student in a lab).

Having a supply of gates in hand, we wanted to build some circuits. We don't do that much hardware hacking, so we hauled down a dog-eared copy of Don Lancaster's *TTL Cookbook* (Howard W. Sams, 1979) and searched it for ideas. The first one we tried was the Set-Reset flipflop, composed of two Nand gates (p. 162). It looks like this, with labels in the first column and expressions in the second:

not-S: 1
not-R: 1
Q: 0 = (*not-S * not-Q*)
not-Q: 0 = (*not-R * Q*)

The feedback reference, or self-reference, or circular reference that is going on between the Q and not-Q cells, is usually a no-no in spreadsheet work. If your calculator rejects these expressions, you are out of luck for the rest of this column. SuperCalc 1.4 accepts them without complaint.

It helps, when operating the S/R flipflop, to turn off automatic recalculation. The manual signal for a recalc (usually the exclamation-mark key) then initiates one cycle of the digital simulation. Set the *not-S* line low by entering a constant 0 into its cell. Recalc; the *Q* cell should go high. Set *not-S* high and recalc; *Q* should remain high. Set *not-R* low and recalc; the flipflop should flip (flop?), making *not-Q* high and *Q* low. Set *not-R* high again and recalc; the outputs should hold their state. And so on.

Making a Clock

The interesting gates are clocked circuits, so we wanted an oscillator expression: a cell that would alternate between 0 and 1 on each recalc. Without it, we would have to enter each clock state manually. We fiddled around for a while, and finally found that the simplest kind of oscillator was the naked self-reference. At least with SuperCalc, if you put into cell A1 the expression

NOT(A1)

you have an oscillator. It changes state, from 0 to 1 or back, each time the sheet is recalculated. From here on, references to *clock* are references to such an oscillator cell. We found it best to have just one oscillator on a sheet, and to refer to its cell wherever the clock signal was needed. It was beginning to dawn on us that, in a spreadsheet Digital Simulator, a cell reference was the equivalent of a wire in the real circuit.

Clocked Flipflops

Now we had a clock, we could make a Clocked S/R flipflop (see Lancaster, page 191). It looks like this:

not-set: 1
not-clr: 1
clk-in: *clock*
clk-set: 0 = (*not-set * clk-in*)
clk-clr: 0 = (*not-clr * clk-in*)
Q: 0 = (*clk-set * not-Q*)
not-Q: 0 = (*clk-clr * Q*)

This circuit is operated much like the unclocked one. You set a low level (constant zero) in the *not-set* or *not-clr* cell when the clock is low, and recalc. Then set a high level (constant one) and recalc again. To enter a constant and then recalc is to simulate applying a signal precisely on the leading or trailing edge of the clock — on

the leading edge if the clock is now 0 and going to 1, on the trailing edge if the clock is now 1 and going to 0.

Moving to the Chip Level

We went on to build a Master-Slave flipflop and a few other circuits out of basic gates, but it was beginning to get tedious. The difficulty comes from the complex mental transformations that must be made between gates on a schematic and the arbitrary row and column addresses of the spreadsheet. It's not difficult to remember that *clock* is A1, but when you have half-entered a circuit of as few as eight gates, it becomes very difficult to keep track of what gate is in what cell. It helps to enter lots of labels, as indicated in the examples here, but it's still easy to miswire the simulation, and hard to debug it once you have.

The named-references and relative-references supported by MultiPlan™ would have been a great help, but we didn't have MultiPlan; we had SuperCalc 1.4, which is only slightly smarter than original VisiCalc. So we tried a different approach.

To this point we have used the spreadsheet to simulate a patchboard and expressions to simulate gates. But nowadays, nobody designs with gates; they design with chips. So why not use the spreadsheet to simulate a wire-wrap circuit board, and build simulated chips? We decided to set up a spreadsheet template for each common chip. Take, for instance, the template for a 7400 Quad 2-input Nand:

```
7400 ---------
 1: A          1
 2: B          1
 3: -A * B     0 = ( A * B )
 4: C          1
 5: D          1
 6: -C * D     0 = ( C * B )
 9: E          1
10: F          1
 8: -E * F     0 = ( E * F )
12: G          1
13: H          1
11: -G * H     0 = ( G * H )
```

The chip is represented by a rectangular patch of the worksheet, two columns wide and with as many rows as it has signal pins. The left-hand column contains labels that document the pin numbers and their uses; the pins are given in logical, not numeric, order. The right-hand column contains the expressions that implement the functions of the chip. Input pins are entered with constant values (usually 1, since TTL inputs float high).

The idea is to copy the chip-patch to wherever on the sheet a chip of this kind is needed. Then you "wire it up" by entering cell references into its input-pin cells and entering the names of its output-pin cells into the inputs of other chip-patches.

Enter the IF

A fine idea, we thought, until we advanced beyond the level of the 7400, 7402, etc., to more elaborate things like a 7474 dual D-type latch. It obviously wouldn't be handy to incorporate two gate-level simulations of D-type latches into the simulation of the chip. First, it would make the chip template too bulky, and second, it would add too many cells to what should be an elementary function (SuperCalc was already slowing down quite noticeably). We wanted to keep the one-row-per-pin relationship in each chip template. Therefore, we wanted a one-cell expression that would simulate *all* the functions of a D-type latch.

We worked out such an expression on paper, based on SuperCalc's IF function. The IF function has a form reminiscent of LISP:

IF (*predicate, true, false*)

The calculator evaluates the expression *predicate*. If its value is nonzero, the value of expression *true* becomes the value of the cell; if zero, the value of *false*.

Here is how the D-type latch is supposed to work (condensed from Lancaster, page 197):

(1) If the D input is positive, the Q output goes or stays positive when the clock goes positive.

(2) If the D input is grounded, the Q output goes or stays grounded when the clock goes positive.

(3) If the Preset and Clear inputs are both positive, they do not interfere with the clocked operations.

(4) If the Preset input is grounded, the flipflop immediately goes to a state with not-Q grounded and Q positive.

(5) If the Clear input is grounded, the flipflop immediately goes to a state with not-Q positive and Q grounded.

And here is how we stated those rules using nested IF functions (with pretty-printing here for clarity):

```
IF( NOT(preset), 1,
  IF( NOT(clear), 0,
    IF( clock, D, self ) ))
```

The reference to *self* is a reference to the cell that contains that whole humongous expression.

We had severe doubts as to whether SuperCalc could handle a triply nested, self-referential IF function, but it did. The resulting 7474 template looks like this:

```
7474 ---------
 1: a,clr      1
 2: a,D        1
 3: a,clk      clock
 4: a,set      1
 5: a,+Q       (the big IF function)
 6: a,-Q       NOT(pin 5 )
13: b,clr      1
12: b,D        1
11: b,clk      clock
10: b,set      1
 9: b,+Q       (again, the big IF )
 8: b,-Q       NOT(pin9 )
```

We then subjected SuperCalc to an even sterner test: the simulation of a JK-type latch. Look up the rules for that one yourself; here is an IF that implements it:

```
IF( NOT(set), 1,
  IF( NOT(clear), 0,
    IF( clock, self,
      IF( J <> K, J,
        IF( J, NOT(self), self ))))
```

This is not the kind of expression that turns up in financial modeling, so we wouldn't have been surprised had SuperCalc tripped over it. However, it worked just fine. Here's a tip of the Intern's sterile cap to Sorcim for not building any arbitrary limits into their product.

We haven't put any of this to practical use, but perhaps a reader can. Even if it doesn't turn out to be useful for real design work, it is certainly a grand educational tool. It's also fun to play with. Here are some problems that we have considered, but not solved.

The most complicated chip for which we've built a template is the 74164 8-bit shift register. That one needed two columns of expressions. One column picked up and shifted the current outputs on the trailing edge of the clock; the other column delivered the new outputs on the leading edge. It's tricky getting both columns to reset properly. Can you do it? What about other chips, like decade counters, 1-of-8 distributors, comparators, and parity generators? Can they be modeled as spreadsheet templates?

More advanced logic circuits need a multiphase clock, in which the master oscillator's beat is divided into phases (like the T states of an 8080). Can you devise an arrangement of cells to yield a four-phase clock cycle? Its heart would be our simple oscillator, but its outputs would be T0, T1, T2, and T3. It ought also to have a numeric display so the operator could easily see what phase it was in, and some kind of instant reset to the just-before-T0 state. Maybe there's a standard chip for this that you could model.

Then there's the problem of handling address and data busses. If the simulator is to be easy to operate, it should be possible to enter a 16-bit address or an eight-bit byte, as a number, to a single cell, with its binary values then somehow distributed automatically over eight or six-

teen simulated bus lines. Conversely, it would be nice to have a simulated LED display cell that would show the value of eight binary output cells as a single value. These problems (especially the problem of decimal-to-hex display) may not be soluble with a general-purpose spreadsheet calculator.

The spreadsheet calculator imposes a particular order of calculation. What, if anything, is the relationship between that and propagation delay in the simulated circuit? Can order of calculation be made useful in the simulation? Does it impose constraints on the realism of the simulation?

Actually, if you were to solve all these problems, you would have the basis of a nice software product. If you market such, don't forget to give credit where it's due. Have fun, and keep those cards and letters coming.

DDJ

Forth and the Motorola 68000

by Joe Barnhart

Joe Barnhart, P.O. Box 4383, Santa Rosa, California 95402.

One of the most powerful microprocessors available, the Motorola 68000 has an architecture that lends itself to fast, efficient, "threaded" languages such as Forth. If you have an interest in the inner workings of the MC68000 processor or are interested in the heart of a threaded interpretive language – the "inner" or "address" interpreter – then read on.

Although classed as a 16-bit microprocessor, the MC68000 maintains full 32-bit registers for all address and data information.[1] With its 24 bits of addressing range, the processor can directly access up to 16Mb of main memory without need of segmentation or bank-switching schemes. But the most useful feature of the MC68000 (for Forth designers) is its rich set of addressing modes and its regular (or consistent) instruction set. It supports 14 different addressing modes, including the all-important "indirect with auto-increment" mode (which is vital to a threaded language).

Any threaded interpretive language can unlock the power of the MC68000 processor, but Forth has characteristics that make it the ideal choice for the language designer. Most of all, its popularity assures a steady supply of programs that can be adapted, with minimal effort, to the new hardware.

Not all Forths are the same, though, and differences that appear subtle from the outside can have profound effects on the overall performance of the language. One such difference is the choice of address interpreter: whether direct or indirect threaded code is used.[2] The performance of the system can be judged by three criteria: speed, compactness, and utility.

Tradeoffs and Performance

Speed

Most people, when asked what they most want in a computer, will say that they want more speed. In a Forth system, people talk about two measures of speed: compilation time, or the time to add definitions to the system, and execution time, or the time to execute those definitions. Compilation time is governed by factors such as I/O speed and dictionary structure. Execution time is governed by the choice of address interpreter and the ratio of assembly definitions to high-level definitions. In this article, I address only execution speed.

Compactness

One of the greatest advantages of a threaded language is its economy of computer memory. A typical Forth system on an 8-bit computer is less than 16Kb in size and includes an editor, assembler, interpreter, compiler, and disk system. If small size is your goal, your system will run somewhat slower than one optimized for speed. Compactness is affected by the ratio of assembly definitions to high-level definitions. The address interpreter also has an impact on the size of the overall system.

Utility

Unfortunately not all factors fall neatly into two piles, one marked "speed" and the other marked "compactness." Some factors influence the usability of the final system. For example, if you want to use programs already written for another computer (such as an IBM PC), you can tailor the architecture of the new Forth system to lessen the conversion effort; it may mean using an indirect threaded interpreter instead of a direct one. Another example is the size of the data and return stacks. Some applications demand greater than 16-bit addressing, which is the traditional size of Forth's stacks. To use a stack wider than 16 bits means the Forth system must be larger.

The Address Interpreter – The Heart of Forth

In threaded languages, programs are represented by a list of addresses. These

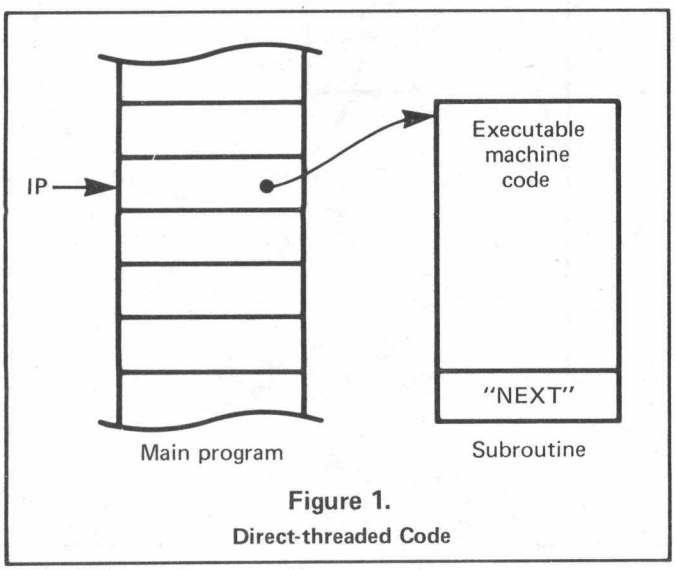


Figure 1.
Direct-threaded Code

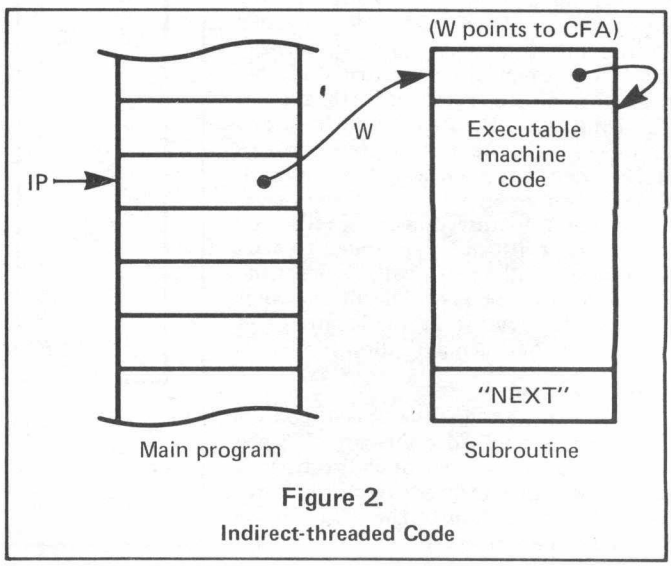


Figure 2.
Indirect-threaded Code

addresses refer in some way to executable machine code. Three methods of implementing threaded code languages are popular: direct, indirect, and token threaded interpreters. In this sense, *interpreter* means the address interpreter – a short (usually less than 10 bytes) machine-coded routine that looks up the address of the next high-level instruction to be executed.

From the language user's standpoint, all three address interpreters may appear the same. As long as the user disregards preconceived ideas about the physical layout of his or her programs, the programs are transportable between systems that use different address interpreters. In a few cases, it is impossible for the user to ignore the physical layout of the programs (i.e., when writing a binary program loader that must relocate the programs after loading them).[3] When possible, Forth programmers can enhance the portability of their programs by using only built-in words that return key addresses, such as LFA, CFA, and PFA in a fig-Forth system.

Direct Threaded

The fastest of the group is the direct-threaded interpreter.[4] In Figure 1 (page 18), the interpreter is calling a subroutine from a main program. In Forth, a subroutine is any procedure (or word) already defined. The IP in the figure is the instruction pointer, a bit of Forth jargon that refers to the pseudo program counter of the threaded interpreter.

With direct-threaded code, the contents of IP point directly to executable machine code. To jump to the code, the processor jumps indirectly via the IP register. To process the next instruction, the microprocessor must first increment the IP register to point to it. Some micros, like the MC68000, accomplish this in one step using an "indirect with auto-increment" addressing mode.

Listing One (page 22) presents the MC68000 version of the direct-threaded address interpreter. At only two instructions, it is a marvelously efficient mechanism for implementing a Forth system. The total time taken to thread from one routine to the next is expressed as 16 clock cycles in the listing. In a typical system with an 8 MHz clock, the instructions take 2 microseconds. Since it's only 4 bytes long, it can be appended to each definition without greatly increasing *memory* requirements. (The alternative is to keep this routine in one location and jump to it from each definition.)

Indirect Threaded

Next fastest in the speed contest is the indirect-threaded interpreter. Since it uses one more level of indirection, it requires another memory fetch to thread from one instruction to the next. This is the type of interpreter used by all "fig-compatible" versions of Forth, because it is somewhat less machine dependent than the direct-threaded version.[5]

As you can see in Figure 2 (page 18), the IP in this machine points to a location called the code field address, or CFA, of the definition to be executed. The contents of the CFA point to executable machine code. To thread from one instruction to the next, the microprocessor uses two indirect memory operations where before it used one. As before, the microprocessor automatically increments the pointers as they are used.

Listing Two (page 24) contains the code for an indirect-threaded address interpreter in MC68000 assembly language. Only one instruction is added to the direct version, at a cost of 2 bytes and 1 microsecond.

Token Threaded

Token-threaded interpreters add yet another level of indirection to the process. I did not implement one in MC68000 instructions, but this is accomplished with one more indirect fetch.

16-Bit Addressing and the MC68000

Most Forth systems are defined with 16-bit data and address elements. An address space of 64Kb is enough for most applications, and 16-bit addresses make Forth a very compact language. Where does the address space begin and end? At zero and $FFFF_H$?

Negative Addresses?

The MC68000 processor allows you to define 32- or 16-bit addresses for any operation. You can mix them at will and interchange data and address registers with impunity. One reason this works is that address registers (A0 ... A7) *sign extend* their operands when they are loaded. That means that an address of 0100_H (16 bits) is sign extended to 00000100_H (32 bits). Fine. The problem is that addresses of 8000_H to $FFFF_H$ are treated as negative addresses and are sign extended to 32 bits by loading the upper 16 bits with $FFFF_H$ (i.e., 8000_H is sign extended to $FFFF8000_H$). The addressing space of the MC68000 lies on a "number circle" as shown in Figure 3 (page 19).

Many MC68000-based computers do not allow the programmer to use the memory region between $FFFF8000_H$ and 00000000_H. In a CP/M-68K computer, for example, the operating system is

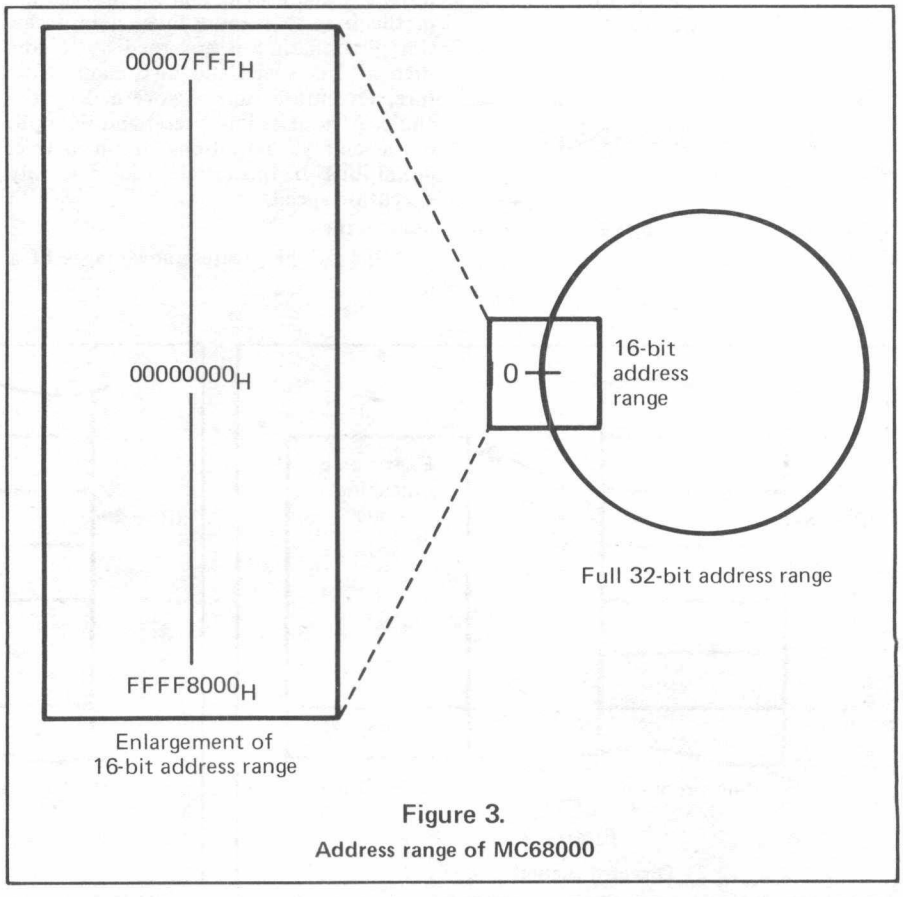


Figure 3.
Address range of MC68000

located either at the top or at the bottom of memory.[6] Obviously a 16-bit version of Forth can't occupy both top and bottom within a CP/M-68K system. Is there any way to circumvent the addressing problem and still have a 16-bit version of Forth?

Address Independent Forth

A solution exists for both direct- and indirect-threaded code interpreters. The solution is to dedicate one register in the MC68000 to hold a base address, or starting address, for the Forth system. All memory references are made relative to the base pointer (or BP). The resulting system can run at any address by simply changing the value of the BP register.

In Listing Three (page 24), both direct- and indirect-threaded interpreters are shown. In the direct-threaded version, the IP register is loaded with the correct 32-bit address at boot-up and thereafter maintains a "real" (32-bit) instead of a "logical" (16-bit plus base pointer) address; it doesn't need to be combined with the BP register. The address of the executable code is combined with the BP register at a cost of 2 bytes and 0.75 microseconds (still an 8 MHz clock). The total time for the direct-threaded address interpreter is now 2.75 microseconds.

The indirect-threaded address interpreter pays a heavier price. It uses the BP register twice at a cost of 2 bytes and 0.75 microseconds per occurrence. The total time for the indirect-threaded address interpreter is 4.5 microseconds.

Even though they run somewhat slower, the address-independent versions of Forth have greater utility than the versions that directly address memory. For instance, the address-independent versions allow use of 64Kb of memory, regardless of where the memory is placed (as long as it is contiguous). In a large address space, you can even keep several separate copies of address-independent Forth, each running its own application.

Timing Comparisons

These Forth systems were implemented on a Sage II computer, a MC68000-based machine with 512Kb of R/W memory and two 320Kb floppy disk drives. Its processor runs at 8 MHz with no memory wait states.

I chose benchmark programs (Listings Four, Five, and Six, pages 24-26) that use memory access functions instead of disk I/O functions. They are programs that I received from Laboratory Microsystems, a vendor of Z80, IBM, CP/M-86, and CP/M-68K Forth systems. Jim Gilbreath's program (Listing Four) allows some comparisons to be made with other languages.[7]

	Byte	Interface Age	Eight queens
Indirect	19.5	9.5	8.5
Direct	15.5	7.5	7.0
Improvement	21%	21%	18%

Table 1.

Two different Forth systems are compared here: an indirect-threaded code version and a direct-threaded code version. Both versions use a base pointer register for memory accesses. Thus these versions of Forth are insensitive to their position in the address space of the MC-68000.

Results

Table 1 (page 20) summarizes the results of the three benchmark programs. Actually the indirect-threaded interpreter didn't do too badly. Instead of the 40% speed advantage you may have expected from the address interpreter times, the direct-threaded code averaged only 20% or so faster. Why? Because the microprocessor spends only part of its time executing the address interpreter. Most of the time is spent executing code that implements the Forth instructions.

Compare the Sieve of Eratosthenes times to those published in *Byte* magazine.[8] The program of Listing Four was run 10 times, as were the programs in *Byte* (Table 2, page 20). As you can see, the Forth systems run slower than native code compilers on this particular benchmark. In all honesty, Forth programs will almost always run slower than those produced by a good native code compiler. Although slower, the Forth system is interactive (much like a BASIC interpreter), which speeds program coding and testing. The native code compilers adhere to the cumbersome cycle of edit, compile, assemble, link, load, and debug for creating programs.

In summary, Forth and the Motorola 68000 microprocessor are made for each other. Used together, they offer a fast, powerful programming environment. The MC68000 processor allows the language designer great flexibility in designing a Forth system.

68000 8 MHz		Assembly	0.49	Andrew Wood
68000 8 MHz		SMPL (Ebnek)	2.6	Steve Keller
68000 8 MHz (Sun PM68K)	ROS	Pascal (Telesoft)	4.28	Craig Maudlin
68000 8 MHz (Sun PM68K)	ROS	Ada (Telesoft)	4.4	Craig Maudlin
68000 Wicat 150WS	MCS/Unix	C (Johnson)	4.71	Authors
68000 8 MHz (HP-9830)	ROS	Ada (Telesoft)	5.0	Craig Maudlin
68000 8 MHz (HP-9830)	ROS	Pascal (Telesoft)	5.0	Craig Maudlin
68000 8 MHz	UCSD	Pascal (Softech native)	5.0	Softech
68000 8 MHz		Pascal (IMS Inc.)	5.8	Steve Keller
68000 8 MHz (HP-9830)	Pascal 1.0	Pascal (Hewlett-Packard)	5.9	Craig Maudlin
68000 Charles River 68	UNOS	C	6.3	Authors
68000 Wicat 150		Pascal	6.5	Richard Lane
68000 (4 MHz)		Pascal (Pascal MT)	9.00	BYTE, Sept. 1981
68000 8 MHz Exormacs		C (Whitesmiths)	9.82	Douglas K. Beck
68000 (4 MHz)		Pascal (Telesoft)	10.2	BYTE, Sept. 1981
68000 8 MHz Exormacs		Pascal (Motorola 1.2)	11.2	Douglas K. Beck
68000 8 MHz	MSP68000	FORTH (Hemenway)	27	Walt Patstone

Table 2.*

**From Eratosthenese Revisited by Jim Gilbreath and Jerry Gilbreath appearing in the January, 1983 issue of* BYTE *magazine. Copyright © 1983 Byte Publications, Inc. Used with the permission of Byte Publications, Inc.*

References

[1] Motorola, 1982, *16-Bit Microprocessor User's Manual,* 3rd ed., Prentice-Hall.

[2] Terry Ritter and Gregory Walker, Sept. 1980, "Varieties of Threaded Code for Language Implementation," *Byte,* pp. 206-227.

[3] Joe Barnhart, Sept. 1982, "A Relocating Loader in Forth," *Dr. Dobb's Journal,* pp. 34-40.

[4] Ritter and Walker, 1980.

[5] Mitch Derick and Linda Baker, 1982, *FORTH Encyclopedia,* 2nd ed., Mountain View Press.

[6] Digital Research, 1983, *CP/M 68K Operating System Programmer's Guide,* Digital Research.

[7] Jim Gilbreath and Jerry Gilbreath, Jan. 1983, "Eratosthenes Revisited," *Byte,* pp. 283-326.

[8] Digital Research, 1983.

DDJ

(Listing begins below)

Glossary of Terms

address interpreter The short machine language program that actually threads, or executes, instructions in a threaded interpretive language. On most microprocessors, the address interpreter is 2-30 bytes long.

BP Base pointer, an address register that holds the 32-bit absolute starting address of the Forth system. All memory references are made relative to this address.

CFA Code field address, a Forth term that denotes a special location in a Forth definition that holds the address of executable machine code. Used in indirect-threaded code implementations.

dictionary In a Forth system, the total collection of "words" (functions and procedures) defined in the system.

direct-threaded code Threaded code in which the addresses point directly to executable machine code rather than other addresses. (See indirect- and token-threaded code.)

fig Forth Interest Group, a collection of Forth enthusiasts responsible for popularizing the language. They have published public-domain versions of Forth for most microprocessors. Most Forths sold today, even those commercially available, are derivatives of fig's original effort.

indirect-threaded code Threaded code in which the addresses point to the address of executable code. Uses one additional level of indirection when compared to direct-threaded code.

inner interpreter (See address interpreter.)

IP Interpretive pointer, the pseudo program counter for a threaded interpretive language. The IP always points to the instruction (represented by an address) to be executed next.

NEXT The traditional Forth name for the entry point into the address interpreter. On some machines, the address interpreter is held in one place and jumped to as needed. Others compile the address interpreter after each machine code routine.

sign extend When converting from a smaller to a larger word size, signed numbers must be sign extended to maintain their correct value. Represented in 16 bits, the decimal number -100 is $FF9C_H$. Sign extended to 32 bits, it becomes $FFFFFF9C_H$. Without sign extension, it would be $0000FF9C_H$, which is 65436 decimal. The bits added to the larger word are set to the value of the sign bit of the original smaller word.

threaded interpretive languages Languages like Forth where the subroutines that comprise programs are represented by their addresses. As a class, they are fast, compact, and naturally interactive.

token-threaded code Threaded code in which the addresses referenced by the IP are entries in a table of "tokens." The table contains the address of the code field (CFA), which contains the address executable code. Uses one additional level of indirection when compared to indirect-threaded code and two additional levels compared to direct-threaded code.

W The "word" pointer in an indirect-threaded interpreter. The IP points to a location that contains W. W, in turn, points to an address that contains executable code.

Listing One (Text begins on page 18)

Note: A0 is the IP register, and A1 is used for scratch.

Instruction	bytes	clocks	description
MOVEA (A0)+,A1	2	8	move (IP) -> scratch
JMP (A1)	2	8	jump (scratch)
Total bytes, cycles:	4	16	

Listing Two

Note: A0 is the IP register, A1 is the W register, and A2 is used for scratch.

Instruction	bytes	clocks	description
MOVEA (A0)+,A1	2	8	move (IP) -> Word reg
MOVEA (A1)+,A2	2	8	move (W) -> scratch
JMP (A2)	2	8	jump (scratch)
Total bytes, cycles:	6	24	

Listing Three

Note: A0 is the IP register, A1 is the base pointer register, and D0 is used for scratch.

Instruction	bytes	clocks	description
MOVE.W (A0)+,D0	2	8	(IP) -> scratch
JMP 0(A1,D0.L)	4	14	jump (BP+scratch)
Total bytes, cycles:	6	22	

Direct threaded interpreter

Note: A0 is the IP register, A1 is the base pointer register, D0 is the W register, and D1 is used for scratch.

Instruction	bytes	clocks	description
MOVE.W (A0)+,D0	2	8	(IP) -> Word reg
MOVE.W 0(A1,D0.W),D1	4	14	(BP+W) -> scratch
JMP 0(A1,D1.W)	4	14	jump (BP+scratch)
Total bytes, cycles:	10	36	

Indirect threaded interpreter

Listing Four

```
Screen # 16
  0 ( Eratosthenes Sieve Prime Number program in FORTH )
  1 ( by Jim Gilbreath, BYTE September 1981 page 190 )
  2 FORTH DEFINITIONS DECIMAL
  3 8190 CONSTANT SIZE      0 VARIABLE FLAGS        SIZE ALLOT
  4
  5 : DO-PRIME      FLAGS SIZE 1 FILL
  6                 0 SIZE 0
```

```
 7                  DO FLAGS I + C@
 8                     IF I DUP + 3 + DUP I +
 9                        BEGIN DUP SIZE <
10                        WHILE 0 OVER FLAGS + C! OVER + REPEAT
11                        DROP DROP 1+
12                     THEN
13                  LOOP
14                  .  ." primes " ;
15 ;S
```

Listing Five

```
Screen # 17
  0 ( Interface Age Benchmark program )
  1 : BENCH DUP 2 / 1+ SWAP ." Starting " CR
  2         1 DO DUP I 1 ROT
  3           2 DO DROP DUP I /MOD
  4              DUP 0= IF DROP DROP 1 LEAVE
  5                ELSE 1 = IF DROP 1
  6                  ELSE DUP 0 > IF DROP 1
  7                    ELSE 0= IF 0 LEAVE
  8                      ENDIF
  9                    ENDIF
 10                  ENDIF
 11                ENDIF
 12              LOOP
 13         IF 4 .R ELSE DROP ENDIF
 14         LOOP DROP CR ." Finished " ;
 15 ( see FORTH DIMENSIONS Vol II, No. 4, page 112 )
```

Listing Six

```
Screen # 29
  0 ( Eight Queens Problem, by Jerry Levan )
  1 ( Exercise in Recursion from                          )
  2 ( FORTH DIMENSIONS II/1 page 6                        )
  3 : 2* DUP + ; ( double a value )
  4 : MYSELF ( allow a word to call itself by recursion  )
  5   LATEST PFA CFA , ; IMMEDIATE
  6 : IARRAY ( makes an array of 1's, as given by input  )
  7   <BUILDS 0 DO 1 , LOOP
  8   DOES> SWAP 2* + ; ( leave indexed address in array )
  9 8 IARRAY A ( these form workspace for solutions       )
 10 16 IARRAY B 16 IARRAY C 8 IARRAY X
 11 : SAFE
 12   SWAP OVER OVER OVER OVER - 7 + C @ >R
 13   + B @ >R DROP A @ R> R> * * ;
 14 -->
 15
```

```
Screen # 30
  0 ( Eight Queens Problem, continued )
  1 : MARK
  2   SWAP OVER OVER OVER OVER - 7 + C 0 SWAP !
  3   + B 0 SWAP ! DROP A 0 SWAP ! ;
  4 : UNMARK
  5   SWAP OVER OVER OVER OVER - 7 + C 1 SWAP !
  6   + B 1 SWAP ! DROP A 1 SWAP ! ;
  7 0 VARIABLE TRIES
  8 : PRINTSOL ." found on try " TRIES @ 6 .R 8 0
  9   DO I X @ 1+ 5 .R LOOP CR ;
 10 : TRY 8 0 DO 1 TRIES +! ?TERMINAL IF QUIT THEN DUP I
 11   SAFE IF DUP I MARK DUP I SWAP X ! DUP 7 <
 12   IF DUP 1+ ?STACK MYSELF ELSE PRINTSOL THEN
 13   DUP I UNMARK THEN LOOP DROP ;
 14 : DO-IT 0 TRIES ! 0 CR TRY ;
 15 DO-IT ;S
```

End Listing

A 68000 Forth Assembler

by Michael A. Perry

Michael A. Perry, Even/Odd Designs, Berkeley, CA 94702.

In this article I will describe the nature of assemblers in Forth, and the use and implementation of a fairly typical example: a Forth assembler for the 68000. I hope to make clear the power of such assemblers, some of the trade-offs involved, and why this assembler was implemented the way it was. For clarity I will ignore implementation dependencies: I will occasionally make statements which are true for my system, but which may vary slightly on other systems.

In brief, a Forth system is an interactive programming environment in which routines (called words) are kept in a data structure called the dictionary. The programmer can add new words which can be defined either in terms of existing words, or in terms of machine code. An assembler in a Forth environment is a tool for defining words using machine code. It is not intended for writing standalone applications in assembler. A Forth cross-assembler is a very similar tool which produces code which will run in a different environment, possibly on a different processor. This article is concerned only with ordinary Forth assemblers. These assemblers are small because they use the existing Forth words, so the size of the assembler is only an increment to the size of the system. For example, this assembler requires about 3 Kbytes, added to a 12 Kbyte system. For comparison, the assembler provided with CP/M-68K requires 44K plus 6K for a symbol table file.

In writing applications, the assembler is rarely used until the algorithms have been tested and debugged in high level code. At the initial stages of a design, the programmer's time is far more valuable than the machine's. When an application is running, it might prove to be too slow. If so, you now decide where the most time is being spent, and rewrite just that routine in code. Repeat this process until the application is fast enough. Avoid writing in code unless necessary: it limits portability. In rare cases, if you have a very time-critical application, you may end up with nearly everything in code. Even then, writing the high level code first will produce results, and the final product, most quickly. Always be prepared to throw out early designs and start over. The key to success is iteration: do it over until you get it right. This is why it is so important to implement a simple version of the application first to find out if your basic idea is workable.

The names of Forth words can be any string of up to 31 ASCII characters excluding the space character. Words in the dictionary are organized into groups called vocabularies. The assembler is a vocabulary named ASSEMBLER. Most words in the assembler are named for the instruction mnemonics of the host processor, in this case the 68000. When such a word is executed, it appends an appropriate sequence of bytes to the dictionary. Other words in the assembler are used for addressing modes, structured conditionals, macros, and possibly other extensions. Given a few constraints, it is possible to implement a very powerful assembler very simply.

The two major constraints are on syntax and forward reference. As is generally true in Forth, forward reference is not allowed. That is, you cannot use anything until it is defined. I believe this is a good thing, but this issue raises endless debate and I will not try to end it here. It is much simpler (and therefore faster in execution) to use an operator final syntax. This means that instructions are written in the form: source destination operation. While unconventional, this format is very flexible and simple to use. An algebraic syntax pre-processor could easily be used if the speed degradation could be tolerated.

The dictionary grows towards high memory as new words are added. Most data structures are also allocated within the dictionary. The variable DP points to the next free address. The word HERE returns the value of DP. The word , (comma) appends a 16-bit value to the dictionary. The word C, (c-comma) appends an 8-bit value (one byte). The assembler is built on comma and c-comma.

Error Control

When I use an assembler, I expect it to have several important characteristics. The first is correctness: good input must result in good output. The second is speed: I want the assembler to produce its output as rapidly as is possible. The third is predictability: when I write assembly code, I will do the optimizing. I will not use an optimizing assembler – I hate surprises. Finally, I prefer to avoid overlaid (or "smart") operators – that is, operators which allow me to be lazy by deciding that this time ADD really means ADDI or ADDQ or whatever. They are slower and less predictable than non-overlaid operators. Because Forth assemblers are extensible, any user can add smart operators on top of the dumb ones if he so desires.

Any amount of error checking can be added to an assembler. Ideally, an assembler should only accept legal input. It can be expensive, however, to provide exhaustive error control, especially in terms of speed. Fortunately, most errors can be caught simply. The cost of checking that the stack depth does not change during a definition (nothing extra is left or consumed) and that the structured conditionals are balanced is very small.

The next level of error detection is to test that only allowed addressing modes are used with each instruction. For a very orthogonal architecture, this is very easy. Unfortunately, the 68000 does not quite qualify, despite advertising claims to the contrary. Even so, many cases can be easily checked. Although I do not ordinarily use them, I have included some words for checking that instructions are given valid addressing modes. ??Dn will abort unless fed a data register direct mode. ??An will abort unless fed an address register direct mode. ??jmp will abort unless fed a valid jump mode. Beyond this level, one is faced with rapidly diminishing returns; much more effort gives only small gains.

Assembler Usage

For a detailed and fairly accurate description of the Motorola MC68000, see the manual.[2] As an example of how the assembler is used, take the definition of the word FILL, which fills an area of memory with a given byte. It is used in the form:

address *length byte* FILL

Notice that FILL takes three parameters from the stack and does not leave any. The definition of FILL is shown in Figure 1 (page 29). The word CODE is a defining word. It creates a header for the new word named FILL and sets its code field to point to its parameter field, and leaves the system in execution state. The assembler does *not* use the Forth compiler, as

is often incorrectly assumed. The header is something like a symbol table entry. The code field of any word points to the code to be executed for that word. Ordinarily, all words with the same parent (defining word) share a code segment in the parent. Words defined by CODE consist of a unique code segment, contained in the body of the word, which is pointed to by the code field of the word. The remaining words in the definition assemble a sequence of bytes into the body of the word.

Assembler opcode mnemonics such as MOVE use the word comma to append numbers in line to the parameter field of the word being defined. If the new word is to return to Forth after executing, its definition will end with NEXT. NEXT is a macro which assembles a jump to Forth's address interpreter. Its definition is

```
: NEXT    (NEXT) #) JMP ;
```

where (NEXT) is the address of the interpreter, #) indicates the "absolute word address" mode, and JMP uses comma to append the proper opcode and address to the word being defined. END-CODE marks the end and does some error checking and housekeeping.

SP is the name of the Forth virtual machine's stack pointer register. The word SP leaves a value on the stack, representing direct-addressing mode using A7 (address register 7). The word A7 has the same effect; both are just constants. The word)+ modifies the value left by SP to indicate the "indirect with auto-increment" addressing mode. How this works will be covered later. The word D0 represents data register 0.

The word MOVE assembles a 68000 move instruction. MOVE requires a pair of arguments, each representing an addressing mode. In this case, the assembled code will move 16 bits from the address pointed to by SP into D0, and increment SP by two. The size of the operation is determined by the contents of the variable SIZE, and defaults to 16-bit at the start of each definition. SIZE is set by BYTE, WORD, and LONG. Notice the difference between WORD in the ASSEMBLER vocabulary and WORD in the FORTH vocabulary. In the FORTH vocabulary, the word *WORD* accepts the next token from the text input stream.

The word CODE leaves the system in the ASSEMBLER vocabulary, so the correct WORD will be found. The word LMOVE was added as a special case of MOVE, because of the register shuffle mentioned earlier. LMOVE always assembles a LONG MOVE, without changing SIZE. Notice the use of DO and LOOP in the assembler. DO is passed a data register which will contain the loop count at execution time. DO passes HERE and the register on to LOOP, which assembles a DBRA back to the DO, using the given register.

Implementation

There are many possible and two major approaches used in writing assemblers in Forth. One method is to use many variables which contain state information, and which are used by the instruction mnemonic words to control assembly, then cleared for use by the next instruction assembling word. A more common and more desirable approach is used here. Almost all information is passed on the stack, which does not need to be initialized.

There are also two major approaches to interpreting the addressing mode information passed to an assembling word. One is to use something equivalent to nested IF – ELSE statements to step through a sequence of choices (conditional branches) of what to do. The other approach, used here, is to have the addressing mode words pass values which can simply be masked and merged (ANDed and ORed) to produce the required machine code values. In either case the resulting numbers are then appended to the dictionary. Logic calculations proceed far faster than decisions as a rule, and so the assembler tends to be much faster when using them.

While reading the following descriptions, refer to the source code listing screens and their shadow (comment) screens (pages 35-42). The key idea to look for here is that a machine code instruction is viewed as a collection of bit fields. These bit fields are specified in the manual for the CPU chip. Some are common to many instructions, like the source and destination, mode and register fields.

As mentioned earlier, instructions which need to know the size of the data to be operated on generally use the variable SIZE. The position of the bit fields which control data size moves around more than any other. Nearly all required values are built into the value of SIZE in each of the three cases: BYTE, WORD, and LONG. Notice that at this point in the code I switched to OCTAL. The 68000 instructions contain many 3-bit fields and can be neatly and most naturally represented as octal digits. I was forced to temporarily forget my bias towards hexadecimal. The bit fields for modes and registers are usually in the pattern shown in Figure 2 below.

In defining the words which specify registers and addressing modes, I indulged in a small trick. I created a multi-defining word, REGS, which uses CONSTANT in a loop to CREATE several related constants. REGS is used only twice: once for the data registers and once for the address registers, which are mode 0 and mode 1:

Mode 0 is data register direct, so D5 is 5005.

Mode 1 is address register direct, so A3 is 3113.

```
CODE FILL     (S adr len val -- )
  SP )+ DO MOVE     ( pop 'val' into DØ )
  SP )+ D1 MOVE     ( pop 'len' into D1 )
  SP )+ D7 MOVE     ( pop 'adr' into D7 whose high half is Ø )
  D7 AØ LMOVE       ( move all of D7 into AØ )
  1 D1 SUBQ         ( decrement D1: DBRA goes to -1, not to Ø )
  D1 DO   DØ AØ )+ BYTE MOVE    LOOP
       ( loop until D1 is -1, each time moving the byte in DØ to
       the address in AØ, and incrementing AØ )
  NEXT       ( compile a jump to 'next' )
END-CODE     ( end definition )
```

Figure 1.
Definition of FILL. An example of the Motorola MC 68000 Assembler.

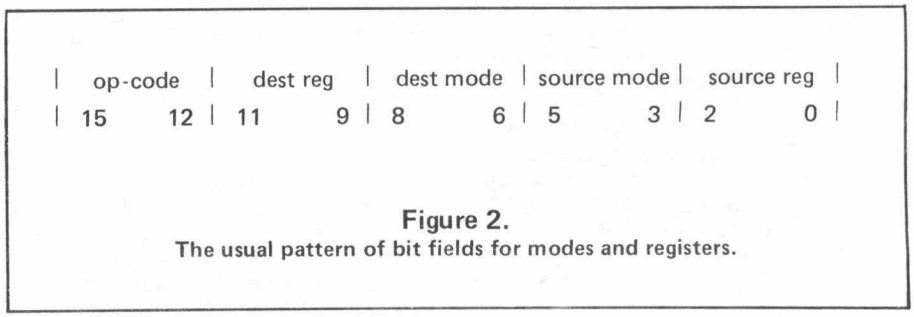


Figure 2.
The usual pattern of bit fields for modes and registers.

Words defined by MODE are used after an address register, and replace the two mode digits (which were 1) with the new mode values. This is done by ANDing off the old values and ORing in the new. They are all address register indirect, with extras.

Mode 2 is address register indirect, so A6) is 6226.

Mode 3 is the same, with post-increment, so A7)+ is 7337.

Mode 4 is the same with pre-decrement, so A7 -) is 7447.

Mode 5 is the same, plus displacement, so 123 A1 D) is 1551, with the displacement value (123) remaining unaffected under the register/mode value on the stack.

Mode 6 is the same, plus an index and a displacement.

123 D4 A1 DI) is 123 under 4004 under 1661.

The remaining possible mode value, mode 7, is used for all remaining modes, which are distinguished by values in the register fields. These modes are defined as constants. #) is 0770, and represents the absolute (16-bit) addressing mode. The name represents "immediate indirect" (think about it).

L#) is 1771, and represents the long absolute (32-bit) addressing mode.

PCD) is 2772, and represents the program counter relative with displacement mode.

123 PCD) is 123 under 2772.

PCDI) is 3773, and represents the program counter relative displaced, indexed mode.

123 D4 PCDI) is 123 under 4004 under 3773.

is 4774, and represents immediate data, 16 or 32 bits.

456 # is 456 under 4774.

Notice that, in every case, from one to three items (16-bit numbers) are left on the stack to represent an addressing mode. The top item usually becomes part of the first 16 bits of an instruction, along with the op-code. Extra items, if any, are assembled following the op-code word.

Certain fields are selected (by masking off the rest) more frequently than others. The word FIELD creates words which select given fields. RS and RD select the source and destination register fields. MS selects the source mode field. The generic term for a completely specified addressing mode is an effective address (EA). The word EAS selects the source effective address, which consists of the source mode and register fields. LOW selects the low 8 bits. The op-code word often contains an EAS field. The word SRC performs OVER EAS OR, which installs that field into the op-code word. The word DST installs the destination register field.

The Forth virtual machine contains five registers. These are assigned to particular 68000 registers. For convenience, either the 68000 or virtual machine register names may be used with the assembler.

Any mode with extra values assembled following the op-code word is said to use extended addressing. The extended addressing modes are handled by six words and a buffer. DOUBLE? leaves a flag which is true if the mode on top of the stack requires that 32 extra bits be assembled. INDEX? looks at a mode, and modifies its extra items into the proper format if it is an indexed mode. MORE? leaves a true flag if the mode has extra items. MORE, appends any extra items to the dictionary (following the op-code word).

Some instructions require two modes, to specify both source and destination addresses. The source mode is specified first, so it is below the destination mode on the stack. Each mode consists of one to three values on the stack. The source mode is used before the destination mode, so it is necessary to move the destination mode values to a buffer until needed. EXTRA? saves any extra items in a buffer named EXTRA, and leaves only the mode value. EXTRA, retrieves such extra items and appends them to the dictionary.

Most of the rest of the assembler consists of the definition and use of defining words which create groups of mnemonics. A couple of examples will suffice. If you are not familiar with defining words in Forth, you should be. That is its most powerful feature.

The word IMM creates words which implement immediate assembler instructions. I will repeat its definition here and go through it in detail.

```
: IMM
    CONSTANT
    DOES> @ >R EXTRA? EAS R> OR
    SZ3 , LONG? ?, ,EXTRA ;
```

Usage: 3000 IMM ADDI

Assembly: n ea ADDI

example: 123 A5) ADDI

Each time IMM is used to define an instruction mnemonic word, it saves a constant value in the definition of that word which distinguishes it from other immediate words. That value is the op-code for the instruction. Immediate instructions contain the following bit fields:

```
| op-code |  size  | mode |  reg  |
| 15   8  | 7   6  | 5  3 | 2  0  |
```

These are followed by 16 or 32 bits of data. When the instruction word is executed, it performs the code following DOES> in IMM, with the address of its own parameter field on the top of the stack. At that address is the constant (op-code) built into it. That value is read by the word @ and saved on the return stack by >R. ADDI is passed the immediate data to use under the mode items for an effective address. EXTRA? saves any extra items, EAS selects the mode and register fields to use, then the op-code is retrieved from the return stack with R> and combined with the EAS by OR. SZ3 installs the appropriate size bits from SIZE, and the word comma appends the op-code word to the dictionary. This leaves only the immediate data on the stack, and LONG? determines whether ?, should append 16 or 32 bits to the dictionary. Finally, the saved extras (if any) are retrieved and appended following the immediate data by ,EXTRA.

Numerous other defining words built along very similar lines are used to define most of the remaining instructions. Many words are in a class by themselves and so are defined by the word colon (:), just like macros. The conditional *instructions* are so regular that I used another trick defining word. SETCLASS repeatedly uses a given defining word, each time with a different argument, to define multiple words at once. It is used to define all 46 conditional instructions by repeating each of the three defining words 16 times. Two junk mnemonics are also created which will not be used. It would be slightly better in this case to define all 46 words separately, but I wanted to show what kinds of things can be done.

Finally we come to the structured conditionals. The following examples will be discussed:

```
            A3 )+ D1 CMP 0<
IF          D0 A7 ) MOVE
ELSE        A7 ) D0 MOVE
THEN
BEGIN       A3 D2 CMP 0=
WHILE       A0 )+ D0 MOVE
REPEAT
```

In the first example the result of a comparison is to affect certain flags in the status register. IF assembles a conditional branch instruction whose op-code (and condition) is specified by 0<, which leaves the value for a jump if greater or equal to zero. ELSE resolves the branch address for the IF and assembles an unconditional branch whose address is resolved by THEN. WHILE assembles a conditional branch to the address following the REPEAT, which assembled an unconditional branch back to the BEGIN.

Notice that there is no need for any labels. A primary source of clutter in ordinary assembler code is the large number of labels with meaningless names used for branch destinations. Also note that the structured conditionals defined here use only one-byte offsets. Because the assembler is one pass, it is necessary to reserve the space for the offset before its

size is known. Since CODE routines in Forth are always rather short, one byte is enough. If it is not, I simply replace these definitions with nearly identical definitions which use only 16-bit offsets.

Finally, it should be noted that there is no need for words to create data structures for use with the assembler. A Forth assembler is part of a Forth environment, and any data structure created by ordinary Forth defining words can be referred to by assembler routines. For example:

```
VARIABLE FOO
CODE BAR FOO #) NEG NEXT
    END-CODE
```

BAR will negate the contents of the variable FOO.

References

1. Leo Brodie, 1981, *Starting Forth*, Englewood Cliffs: Prentice Hall.
2. Motorola, 1982, *16-Bit Microprocessor User's Manual*, 3rd. ed. Prentice-Hall.
3. C. H. Ting, *Systems Guide to Fig-Forth*.
4. *Forth Dimensions*, published by the Forth Interest Group, P. O. Box 1105, San Carlos, CA 94070.

(Listing begins below)

A 68000 Forth Assembler (Text begins on page 28)

```
   55
 0 \ 68000 Assembler                                  06Jun83map Assembler
 1 VOCABULARY ASSEMBLER IMMEDIATE   ASSEMBLER DEFINITIONS
 2 ( ' ASSEMBLER CFA  ' ;CODE 10 + !  )
 3
 4 : END-CODE   (S -- )   CURRENT @ CONTEXT ! ;
 5 : $SWAP   SWAP ;
 6 : ?,   (S [n2] n1 f -- )   IF , THEN , ;
 7
 8 1 14 +THRU
 9 FORTH DEFINITIONS
10 : LABEL   (S -- )
11    CREATE   [COMPILE] ASSEMBLER   ASSEMBLER WORD ;
12 : CODE    (S -- )
13    LABEL   HERE  DUP 2- ! ;

   56
 0 \ Sizes                                            31May83map Assembler
 1 OCTAL
 2
 3 VARIABLE SIZE
 4 : BYTE   (S -- )   10000 SIZE ! ;
 5 : WORD   (S -- )   30100 SIZE ! ; WORD
 6 : LONG   (S -- )   24600 SIZE ! ;
 7 : SZ     (S n -- )    CONSTANT DOES> @ SIZE @ AND OR ;
 8 00300 SZ SZ3            00400 SZ SZ4
 9 04000 SZ SZ40           30000 SZ SZ300
10 : LONG?   (S -- f )   SIZE @ 24600 = ;
11 : -SZ1    (S op -- op' )   LONG? IF  100 OR  THEN ;
12

   57
 0 \ addressing modes                                 06Jun83map Assembler
 1 : REGS   (S n -- )   ( register direct )
 2    10 0 DO  DUP 1001 I * OR  CONSTANT  LOOP  DROP ;
 3 : MODE   (S n -- )   CONSTANT  DOES> @ SWAP 7007 AND OR ;
```

```
  This code is for systems compatible with Starting Forth,
which is very nearly 79-Standard.
  This is a structured macro assembler which uses Motorola
mnemonics ( with few exceptions) and the following syntax:
source destination operation. For example,
D0 D2 MOVE   means copy the contents of D0 into D2,
1 # SP -) MOVE   means push a 1 onto the parameter stack.

(S  means 'stack comment' and is equivalent to (
\   means 'comment to end of line'

LABEL is used to name the entry point to a subroutine.
CODE creates a Forth word which will be defined in assembler.
```

```
Many instructions in the 68000 can operate on 8, 16, or 32 bit
data.  Rather than specify the size individually for each
instruction, the variable SIZE contains the size information
for any instruction which needs it. SIZE is set by BYTE, WORD,
and LONG.

SZ defines words which select certain bits from SIZE and install
them into the instruction being assembled. The size field moves
around considerably, and is even inverted in some cases.
?LONG returns a true flag if SIZE is LONG.
-SZ1 is used in a special case where the size flag has the
opposite sense of all other size flags. Good job, Motorola!
```

```
This screen defines the 16 data and address registers, and
the various addressing modes.
Notice that REGS defines several words each time it is used;
```

(Continued on page 36)

A 68000 Forth Assembler (Listing continued, text begins on page 28)

```
 4 0000 REGS     D0   D1   D2   D3   D4   D5   D6   D7
 5 0110 REGS     A0   A1   A2   A3   A4   A5   A6   A7
 6 0220 MODE     )          ( address register indirect )
 7 0330 MODE     )+         ( adr reg ind post-increment )
 8 0440 MODE     -)         ( adr reg ind pre-decrement )
 9 0550 MODE     D)         ( adr reg ind displaced )
10 0660 MODE     DI)        ( adr reg ind displaced indexed )
11 0770 CONSTANT #)         ( immediate address )
12 1771 CONSTANT L#)        ( immediate long address )
13 2772 CONSTANT PCD)       ( PC displaced )
14 3773 CONSTANT PCDI)      ( PC displaced indexed )
15 4774 CONSTANT #          ( immediate data )
```

it is a DO CONSTANT LOOP.

Words defined by MODE are modifiers, they follow an address register name. For example, A5) means address register 5 indirect. Examples of usage follow:

A5)+ A5 -) A5 indirect, post-increment and pre-decrement.
3 A5 D) A5 indirect with displacement of 3.
3 D2 A5 DI) A5 indirect indexed by D2 with displacement of 3.
1234 #) absolute address 00001234 (immediate indirect)
1234.5678 L#) absolute address 12345678 (long imm. indirect).
3 PCD) program counter offset by 3.
3 D2 PCDI) program counter indexed by D2, offset by 3.
1234 # immediate data 1234. (size determined by SIZE).

```
   58
 0 \ fields and register assignments                    06Jun83map
 1 : FIELD   (S n -- )   CONSTANT  DOES> @ AND ;
 2 7000 FIELD RD           0007 FIELD RS
 3 0070 FIELD MS           0077 FIELD EAS
 4 0377 FIELD LOW
 5 : DN?   (S ea -- ea flag )   DUP MS 0= ;
 6 : SRC   (S ea n -- ea n' )   OVER EAS OR ;
 7 : DST   (S dn n -- n' )      SWAP RD  OR ;
 8 A7 CONSTANT SP          A6 CONSTANT RP
 9 A5 CONSTANT IP          A4 CONSTANT W
10 A3 CONSTANT UP
11 : ?MODE   (S f -- )   0= ABORT" Bad Mode"  ;
12 : ??Dn   (S mode -- mode )   DN? ?MODE  ;
13 : ??An   (S mode -- mode )   DUP MS 1 = ?MODE  ;
14 : ??JMP  (S mode -- mode )   DUP MS DUP 2 = SWAP 4 > OR
15    OVER 74 = NOT AND ?MODE  ;
```

FIELD creates words which mask off various bit fields.
RS and RD select the source or destination register fields.
MS selects the source mode field. EAS selects the source effective address field. LOW selects the low byte.
DN? tests for data register direct mode.
SRC and DST contain phrases which occur frequently in source or destination mode calculations.
These are the virtual machine register assignments. The Forth virtual machine has five registers. Each of these is assigned to a 68000 address register. These definitions allow your code to reference the virtual machine registers symbollically. For example, RP)+ SP -) MOVE pops the top item off the return stack and pushes it onto the data stack.

?MODE aborts if passed a false flag. ??Dn requires the data register direct mode. ??An requires address register direct. ??JMP requires any mode legal for the JMP instruction.

```
   59
 0 \ extended addressing                                31May83map Assembler
 1 : DOUBLE?  (S mode -- flag )
 2    DUP L#) = SWAP # = LONG? AND OR ;
 3 : INDEX?   (S (n) mode -- (m) mode )
 4    DUP >R  DUP 0770 AND A0 DI) =  SWAP PCDI) =  OR
 5    IF   DUP RD 10 * SWAP MS  IF  100000 OR  THEN
 6         SZ40 SWAP LOW OR
 7    THEN  R> ;
 8
 9 : MORE?   (S ea -- ea flag )  DUP MS 0040 > ;
10 : ,MORE   (S ea -- )   MORE?
11    IF  INDEX?  DOUBLE?  ?,  ELSE  DROP  THEN ;
12
```

Many of the 68000's addressing modes require additional bytes following the opcode.
DOUBLE? leaves true if the given mode requires 32 bits of additional addressing information.
INDEX? does nothing unless the given mode is an indexed mode, in which case it takes the data under the mode and packs it into the appropriate format.

MORE? returns a true flag if the mode has extended addressing.
,MORE will append the extended addressing bytes to the dictionary.

```
   60
 0 \ extended addressing  extras                        31May83map Assembler
 1 CREATE EXTRA   HERE 5 ALLOT 5 ERASE \ temporary storage area
 2
 3 : EXTRA?   (S (n) mode -- mode )   MORE?
 4    IF  >R  R@ INDEX?  DOUBLE?  EXTRA 1+ SWAP
 5        IF  2! 2  ELSE  ! 1 THEN  EXTRA C!  R>
 6    ELSE   0 EXTRA !
```

This is a holding area for destination extensions.

EXTRA? tests the mode for extensions, and if present stores them at EXTRA.
,EXTRA removes the extras (if any) from EXTRA and adds them to the dictionary.

(Continued on page 38)

```
 7     THEN  ;
 8 : ,EXTRA   (S -- )   EXTRA C@  ?DUP
 9    IF   EXTRA 1+ SWAP 1 =
10         IF  @ ,  ELSE  2@ 2,  THEN  EXTRA 5 ERASE
11    THEN ;
12
13
```

```
   61
 0 \ immediates & address register specific              31May83map Assembler
 1 : IMM   (S n ea -- )   CONSTANT                                  This screen sets the pattern for the rest of the assembler.
 2    DOES> @ >R EXTRA? EAS R> OR SZ3 , LONG? ?,  ,EXTRA ;          Defining words are created, then used to define a group of
 3 0000 IMM ORI            1000 IMM ANDI                            similar instructions. The stack comment (S n ea )
 4 2000 IMM SUBI           3000 IMM ADDI                            indicates the allowed addressing modes. In this case, for
 5 5000 IMM EORI           6000 IMM CMPI                            immediate instructions, a number followed by any effective
 6 : IMMSR   (S n ea -- )   CONSTANT  DOES> @ SZ3 2, ;              addressing mode is required before the instruction. For
 7 001074 IMMSR ANDI>SR    005074 IMMSR EORI>SR                     example  123 A5 ) ORI  means to perform a logical OR of the
 8 000074 IMMSR ORI>SR                                              immediate data 123 with the contents of the address pointed
 9 : IQ   (S n ea -- )   CONSTANT                                   to by address register 5, with the result going to that address
10    DOES> @ >R EXTRA? EAS SWAP RS 1000 * OR R> OR SZ3 , ,EXTRA ;
11 050000 IQ ADDQ          050400 IQ SUBQ
12 : IEAA   (S ea An -- )   CONSTANT DOES> @ DST SRC SZ4 , ,MORE ;
13 150300 IEAA ADDA        130300 IEAA CMPA
14 040700 IEAA LEA         110300 IEAA SUBA
15
```

```
   62
 0 \ shifts, rotates, and bit manipulation               06Jun83map Assembler
 1 : ISR   (S Dm Dn | m # Dn | ea -- )   CONSTANT  DOES> @ >R DN?   ISR creates instructions which shift or rotate.
 2    IF  SWAP DN? IF  R> 40 OR >R  ELSE DROP SWAP 1000 * THEN
 3        RD SWAP RS OR R> OR 160000 OR SZ3 ,
 4    ELSE  DUP EAS 300 OR R@ 400 AND OR R> 70 AND 100 * OR
 5          160000 OR , ,MORE
 6    THEN ;
 7 400 ISR ASL              000 ISR ASR
 8 410 ISR LSL              010 ISR LSR
 9 420 ISR ROXL             020 ISR ROXR
10 430 ISR ROL              030 ISR ROR
11 : IBIT   (S ea Dn | ea n # )   CONSTANT  DOES> @ >R  EXTRA?  DN? IBIT creates instructions which manipulate bits.
12    IF  RD SRC 400  ELSE  DROP DUP EAS 4000  THEN
13    OR R> OR , ,EXTRA ,MORE ;
14 000 IBIT BTST           100 IBIT BCHG
15 200 IBIT BCLR           300 IBIT BSET
```

```
   63
 0 \ branch, loop, and set conditionals                  31May83map Conditionals
 1 : SETCLASS    ' SWAP 0 DO I OVER EXECUTE LOOP DROP ;              There are three classes of conditional instructions:
 2 : IBRA   (S adr -- )   400 * 060000 OR CONSTANT                 branch, decrement and branch, and set. In each case there
 3    DOES> @ SWAP HERE 2+ - DUP ABS 200 <                         is a four bit field which contains the condition code; this
 4    IF  LOW OR ,  ELSE  SWAP 2,  THEN  ;                         field is the only difference between members of a class.
 5 : IDBR   (S adr Dn -- )   400 * 050310 OR CONSTANT              Rather than explicitly define sixteen separate words for each
 6    DOES> @ SWAP RS OR , HERE - , ;                              class, the word SETCLASS is used to define all sixteen at once
```

(Continued on page 40)

A 68000 Forth Assembler (Listing continued, text begins on page 28)

```
 7 : ISET   (S ea -- )   400 * 050300 OR CONSTANT
 8   DOES> @ SRC , ,MORE  ;
 9 20 SETCLASS IBRA     BRA BSR BHI BLS BCC BCS BNE BEQ
10                      BVC BVS BPL BMI BGE BLT BGT BLE
11 20 SETCLASS IDBR     DXIT DBRA DBHI DBLS DBCC DBCS DBNE DBEQ
12                      DBVC DBVS DBPL DBMI DBGE DBLT DBGT DBLE
13 20 SETCLASS ISET     SET SNO SHI SLS SCC SCS SNE SEQ
14                      SVC SVS SPL SMI SGE SLT SGT SLE
15
```

by re-executing the same defining word with a different value for the condition code field each time. Of the 48 words so defined, only DXIT and SNO are not actually useful.

```
   64
 0 \ moves                                          31May83map Assembler
 1 : MOVE   (S ea ea -- )
 2   EXTRA? 7700 AND SRC SZ300 ,  ,MORE ,EXTRA ;
 3 : MOVEQ   (S # Dn -- )   RD SWAP LOW OR 070000 OR , ;
 4 : MOVE>USP   (S An -- )   RS 047140 OR , ;
 5 : MOVE<USP   (S An -- )   RS 047150 OR , ;
 6 : MOVEM>   (S n ea -- )
 7   EXTRA? EAS 044200 OR -SZ1 , , ,EXTRA ;
 8 : MOVEM<   (S n ea -- )
 9   EXTRA? EAS 046200 OR -SZ1 , , ,EXTRA ;
10 : MOVEP   (S Dm d An | d An Dm -- )
11   DN? IF     RD SWAP RS OR 410 OR
12       ELSE   RS ROT RD OR 610 OR
13       THEN   -SZ1  2, ;
14 : LMOVE      7700 AND SWAP EAS OR 20000 OR , ;
15  ( long reg move )
```

To keep the assembler simple, some compromise was made here on compatability with Motorola's mnemonics. The instruction names explicitly indicate the direction of data flow in several instructions. For example,
HEX FFFF SP -) MOVEM> will save all registers on the stack.
(pronounced MOVEM-OUT)
LMOVE is a convenience to compensate for the painful and unneccessary sign extension of address registers. It assembles a long move without affecting SIZE. It is used primarily for moving an entire data register whose high half is zero into an address register. This shuffle is neccessary when loading a 16 bit address into an address register.

```
   65
 0 \ odds and ends                                  31May83map odds and ends
 1 : CMPM   (S An@+ Am@+ -- )   RD SWAP RS OR 130410 OR SZ3 , ;     CMPM
 2 : EXG    (S Rn Rm -- )                                          EXG
 3   DN? IF   SWAP DN?  IF  140500 ELSE 140610 THEN >R
 4       ELSE SWAP DN?  IF  140610 ELSE 140510 THEN >R SWAP
 5       THEN  RS DST R> OR , ;
 6 : EXT    (S Dn -- )   RS 044200 OR -SZ1 , ;                     EXT
 7 : SWAP   (S Dn -- )   RS 044100 OR , ;                          SWAP
 8 : STOP   (S n -- )    47162 2, ;                                STOP
 9 : TRAP   (S n -- )    17 AND 47100 OR , ;                       TRAP
10 : LINK   (S n An -- )   RS 047120 OR 2, ;                       LINK
11 : UNLK   (S An -- )     RS 047130 OR , ;                        UNLK
```

```
   66
 0 \ arithmetic and logic                           06Jun83map arithmetic and logic
 1 : EOR   (S Dn ea -- )   EXTRA? EAS DST SZ3 130400 OR , ,EXTRA ;  EOR  is a special case.
 2 : IDD   (S Dn Dm | An@- Am@- -- )   CONSTANT                     IDD  instructions which take either a pair of data registers,
 3   DOES> @ DST OVER RS OR #SWAP MS IF  10 OR  THEN  ,  ;            or a pair of address registers used in post-increment mode.
 4 140400 IDD ABCD          100400 IDD SBCD
 5 150400 IDD ADDX          110400 IDD SUBX
 6 : IDEA   (S ea Dn | Dn ea -- )   CONSTANT  DOES> @ >R DN?        IDEA  instructions which take an EA and a data register, in
 7    IF  RD SRC R> OR SZ3 , ,MORE                                    either order.
 8    ELSE  EXTRA? EAS DST 400 OR R> OR SZ3 , ,EXTRA  THEN ;
 9 150000 IDEA ADD          110000 IDEA SUB
10 140000 IDEA AND          100000 IDEA OR
```

```
11 : IEAD   (S ea Dn -- )   CONSTANT  DOES> @ DST SRC  , ,MORE ;    IEAD  instructions which take any EA for the source, and a data
12 040600 IEAD CHK                                                    register for the destination.
13 100300 IEAD DIVU         100700 IEAD DIVS
14 140300 IEAD MULU         140700 IEAD MULS
15 : CMP    (S ea Dn -- )   130000 DST SRC SZ3 , ,MORE ;            CMP  another special case.
```

```
   67
 0 \ arithmetic and control                          31May83map arithmetic and control
 1 : IEA   (S ea -- )   CONSTANT  DOES> @ SRC , ,MORE ;           IEA  instructions which take only an effective address.
 2 047200 IEA JSR          047300 IEA JMP
 3 042300 IEA MOVE>CCR
 4 040300 IEA MOVE<SR      043300 IEA MOVE>SR
 5 044000 IEA NBCD         044100 IEA PEA
 6 045300 IEA TAS
 7 : IEAS   (S ea -- )   CONSTANT  DOES> @ SRC SZ3 , ,MORE ;       IEAS  instructions which take only an effective address
 8 041000 IEAS CLR         043000 IEAS NOT                           and use SIZE.
 9 042000 IEAS NEG         040000 IEAS NEGX
10 045000 IEAS TST
11 : ICON   (S -- )   CONSTANT  DOES> @  , ;                       ICON  instructions which assemble a constant value.
12 47160 ICON RESET       47161 ICON NOP
13 47163 ICON RTE         47165 ICON RTS
14 47166 ICON TRAPV       47167 ICON RTR
15
```

```
   68
 0 \ structured conditionals  +/- 256 bytes           01Sep82map Assembler
 1 : THEN   HERE OVER 2+ - $SWAP 1+ C! ;   : ENDIF   THEN ;      The usual complement of Forth conditionals is provided,
 2 : IF      , HERE 2- ;            HEX                         including DO LOOP. The last is unusual in assemblers. No
 3 : ELSE    6000 IF  $SWAP THEN ;                              error checking is provided for mismatched conditionals as
 4 : BEGIN   HERE ;                                             it is desirable when writing code to be able to easily bend
 5 : UNTIL   , HERE - HERE 1- C! ;                              the rules of strictly structured code. Keep your code routines
 6 : AGAIN   6000 UNTIL ;                                       small! It will not only make your code more comprehensible,
 7 : WHILE   IF ;                                               but also make it less error prone.
 8 : REPEAT  $SWAP AGAIN THEN ;
 9 : DO      HERE $SWAP ;                                       Note that any DBcc can replace LOOP.
10 : LOOP    DBRA ;
11 6600 CONSTANT 0=   6700 CONSTANT 0<>
12 6A00 CONSTANT 0<   6B00 CONSTANT 0>=
13 6C00 CONSTANT <    6D00 CONSTANT >=
14 6E00 CONSTANT <=   6F00 CONSTANT >
15 DECIMAL
```

```
  186
 0 $
 1                                                    ********************************
 2                                                    *                              *
 3                                                    *     Michael A. Perry         *
 4                                                    *     1446 Stannage Avenue     *
 5                                                    *     Berkeley, California     *
 6                                                    *     94702                    *
 7                                                    *                              *
 8                                                    ********************************
 9
10
11
12
```

End Listing

Nondeterministic Control Words in Forth

by L. L. Odette

Programmers usually write programs with the expectation that output will depend only on input and on the sequence of program statements, so it might seem surprising at first that making the program counter a random number generator could have other than disastrous consequences. But, in fact, the technique is both interesting and useful.

Of course, to ensure some discipline in the system, we'll let the program counter be a random number generator only at certain strategic parts of the program. The implementation of a coin-tossing controller is called, appropriately, a nondeterministic control structure. This article will explore the implementation and use of nondeterministic control structures in Forth. This work is motivated largely by Henderson's discussion of the use and implementation of nondeterministic control in a variant of LISP.[1]

The main reason for introducing new data or control structures into a language is to increase the expressiveness of the language: Some algorithms are more naturally programmed using one control structure than they are using others. A good example of this is the use of recursion, in place of iteration, where the problem to be solved lends itself to progressive decomposition into similar but simpler problems. Factors such as efficiency and the availability of temporary storage may influence the ultimate choice of a programming technique, but the expressive power of recursive control structures lies in making the solution itself simple and natural.

Just as recursive control is most naturally used in the solutions of a particular class of problems, nondeterministic control leads to simple and natural solutions in another class of problems, those that employ backtracking. As will be demonstrated, nondeterministic control words in Forth can be used to implicitly encode the backtracking process. The result is a substantial gain in the ease of implementation of backtracking algorithms.

The nondeterministic control structures described here also are useful for programming probabilistic algorithms.[2] A probabilistic algorithm uses random numbers to determine branchings in an otherwise deterministic program and may not necessarily require backtracking.

Rabin's systematic study of probabilistic algorithms showed that in certain cases the use of the nondeterministic approach could dramatically speed up computation.[3] For example, the problem of determining whether a natural number N is prime becomes impractical to solve by conventional deterministic methods when N gets large ($N \approx 10^{60}$). Rabin's probabilistic algorithm recognized $2^{400} - 593$ as prime in a few minutes of CPU time on a medium-sized computer.

Probabilistic and Backtracking Algorithms

The idea behind our nondeterministic control structures is simply to toss a coin at the chosen control point to determine which of the next two Forth words is executed. The control word can be viewed as a branch point in the program; the coin toss determines which branch the program executes.

The probabilistic Forth control word ONEOF has the following syntax:

```
ONEOF WORD1 WORD2
```

When ONEOF is executed, either WORD1 or WORD2 is executed next, the choice being made randomly. After execution of the chosen word, control passes to the word following WORD2. ONEOF is defined in screen 33 in Listing One (page 52).

As an example of the use of ONEOF, consider the following Forth construct designed to take an integer N on the stack and to return an arbitrary integer between 0 and N:

```
: RANDOMN DUP IF 1- ONEOF MYSELF 1+ THEN ;
```

MYSELF is the recursive control word used by LeVan.[4] Execution of RANDOMN terminates if N is zero. Otherwise, by a toss of the coin, either N is returned as the choice or RANDOMN is called recursively with N-1 on the stack. You might compare RANDOMN with CHOOSE, the coin tosser used in the definition of ONEOF.

The advantage of nondeterministic control in this implementation of a random number generator lies in the ability to think in terms of an arbitrary (random) choice without having to worry about the details. This is the increase in expressiveness we're looking for in a new control structure. For a further illustration of the utility of nondeterminism, consider the problem of generating an arbitrary permutation of a list and note the natural way PERMUTE solves the problem (see Figure 1, page 48).

As an example of a backtracking program, consider the word RESTRICT that solves the problem of generating an arbitrary permutation of a list that has predefined properties (we may want the elements of the list to have a particular order, or we may require successive elements in the list to be relatively prime). For our definition of RESTRICT, we need to assume we have a word ?PROPERTY that tests the list on the data stack for the particular property of interest, returning a flag on the stack indicating the result of the test. Since we have no guarantee that *any* permutation has the necessary property, we need to supplement our control structures to account for the possibility that a sequence of choices made during successive execution of ONEOF might result in a list without the required property. To do this, we add the backtracking control word NONE. NONE is defined in screen 33 in Listing One.

Execution of NONE restores the program to the state it was in just before execution of the last ONEOF and then forces execution of the word that wasn't executed before. If during execution of the entire program a sequence of choices (by ONEOF) avoids execution of NONE, we can be sure that sequence of choices will be made.

To program for restricted permutations, we need only define RESTRICT as:

```
: RESTRICT PERMUTE ?PROPERTY IF 1 ELSE NONE 0 THEN ;
```

As before, RESTRICT expects the list that will be permuted to be on the stack. By the definition of NONE, if *any* permutations have the required property, then one such permutation will be returned.

Here again is the expressive power of the ONEOF...NONE

L. L. Odette, Boston University, College of Engineering, Boston, Massachusetts 02215.

control structure: We can think in terms of arbitrary choice, knowing that a sequence of choices leading to the desired result will be made if such a sequence of choices can be made at all.

One way of describing the use of NONE in conjunction with ONEOF is to say that execution of ONEOF results in a bifurcation of the Forth program and, in effect, creates two Forth programs that are executed in parallel. If during execution of either program the word NONE is executed, execution of that program is terminated. A program executes successfully if it manages to avoid executing the word NONE.[5]

Alternatively, nondeterministic control can be viewed as a form of multitasking, with ONEOF randomly choosing which of two tasks to run and NONE rescheduling should the running task fail to complete.

Implementation

To implement ONEOF, provision must be made for storing the state of the Forth machine at the time ONEOF is executed. In this implementation, the state of the Forth machine consists of the contents of the return and data stacks.

When ONEOF is executed, the top of the return stack is incremented by four to point after the next two words. One of the two words following ONEOF is arbitrarily chosen for execution, and a pointer to the other is saved on the return stack. Before proceeding, the contents of the data and return stacks are each moved to the dictionary (by DATA> and RETURN>, respectively) in the format $S_N, \ldots S_0, N$ where N is the stack depth and S_N is the top stack entry. The stack depth is the last entry in the dictionary. ONEOF thus saves all the information necessary to backtrack and proceed with execution down the other branch of the program should the word NONE be encountered.

Note that literals are not allowed as the arguments (two following words) of ONEOF. In addition, variables must be used with care when employing nondeterministic control, since the values of variables are not saved and therefore will not be restored if NONE is encountered.

ONEOF and NONE resemble CALL and RET instructions, respectively, the difference being that ONEOF is a CALL to a routine chosen at random from two possibilities, and NONE is either a RET followed by a call to the other routine (the one not chosen on the last toss of the coin) or a RET to an even higher level of the main program if both choices result in execution of NONE. The nondeterministic control structure implemented with ONEOF can be nested arbitrarily deep (limits on dictionary space will ultimately limit the nesting depth). It is worthwhile to note that control structures may be nested to any depth in recursive calls and still transfer control in the correct manner on execution of NONE (see Figure 1). This is because NONE discards the return stack entirely, recreating it exactly as it existed prior to the last execution of ONEOF.

The result of executing a word that uses nondeterministic control represents one instance of all the possible ways that word may have executed. The result of program execution can be described only by a probability density function. If more than one solution (output) is possible, then it's guaranteed that sometimes you'll get one right answer and sometimes you'll get a different right answer. NONE can also be used to force execution of other instances of a word, as will be shown.

Some practical considerations: The word NEW is used to reclaim the dictionary space used by ONEOF for storage of the machine state; 0 is enclosed in the dictionary as a marker. NONE gives an error message if it finds the marker when it tries to restore the stacks.

Example: N Queens Problem

As a final example of the use of a nondeterministic control structure in implementing backtracking algorithms, consider the N Queens problem. The goal is to determine the placement of N queens on an N x N chessboard so that no two queens are in a position to attack each other. LeVan published a solution to the 8 Queens problem to illustrate recursion in Forth.[6] The solution to the N Queens problem (See Figure 2 on page 50) illustrates the use of nondeterministic primitives in backtracking algorithms. LeVan's program can be used for comparison.

The algorithm is straightforward: A queen is placed in each of N rows in turn so long as no two queens can attack each other. If it is impossible to place a new queen so that it is not attacked by other queens already on the board, then the program backtracks to try a different placement of the most recently placed queen. Nondeterminism is introduced by the use of RANDOMN to try a placement in the current row. NONE is executed if a placement would imply attack by one of the queens already placed. ONEOF and NONE are implemented so that, if RANDOMN leads to placement in row j and this in turn results in execution of NONE (because a queen placed in row j can be attacked), row j + 1 is tried for the placement of the queen. NONE thus results in movement to the right for the most recently placed queen that can be so moved and then forces the placement of all subsequent queens to be recomputed.

The word QUEENSOLN returns a list of the row numbers on which the queens are placed, for each column from 1 to N. The predicate ?ATTACKS is the word most involved in the definition of QUEENSOLN. Its job is to look at the current placement of the queens and to decide whether the next placement implies attack by one of the queens already placed. All information used by ?ATTACKS is on the data stack in the form $R_1, \ldots R_N, N, R_{N+1}$ where N is the number of queens placed on the board and R_j is the row placement on column j of the board. For each queen already placed, ?ATTACKS first tests to ensure that placement of the new queen is not in the same row. Possible attack along a diagonal is tested by comparing the sums and differences of the row-column pairs for each placement.

Remember that the list returned by QUEENSOLN is only one of the possible solutions. By executing NONE after completion of the program, another solution can be computed and displayed. Figure 2 gives examples of solutions returned by QUEENSOLN to both the 8 Queens problem and the 12 Queens problem. QUEENSOLN has also been used to find a solution to the 14 Queens problem (4 9 13 6 2 7 12 1 3 5 10 8 11 14).

Discussion

QUEENSOLN supports the claim that nondeterministic control structures increase the expressive power of the Forth language. Compared to the explicit programming of backtracking used by LeVan to solve the 8 Queens problem, use of nondeterministic control yields a simpler program and a more natural expression of the algorithm. What is gained with nondeterminism is the ability to easily and naturally express a trial-and-error process. When programming, we can concentrate more on the properties a solution must have and less on the details of transferring control to reach that solution. We can write the predicates and then think in terms of arbitrary choices, secure that the correct sequences of choices will eventually be made if such a sequence exists at all. The facility for thinking this way is particularly useful when programming searches.

Keep in mind that the use of nondeterminism for backtracking in the N Queens problem is very straightforward. In more complex problems involving many more sources of nondeterminism, a direct formulation in terms of nondeter-

(Continued on page 51)

```
SCR #21
  0 ( L. L. Odette     6-2-83                          MVP-FORTH )
  1 ( Figure 1, Screen 1 of 2: INSERT and PERMUTE words )
  2   VARIABLE LIST-LENGTH
  3
  4 : SHIFT     LIST-LENGTH @ ROLL ; ( Rotate list on stack )
  5
  6 : INVERT    DUP 2+ LIST-LENGTH ! ( Initialize data length )
  7             SHIFT SHIFT SWAP     ( Rotate last, latest to top )
  8             3 PICK    0          ( Set loop limits )
  9             DO SHIFT LOOP ;      ( Last and latest reversed )
 10
 11 : +INVERT   1+ INVERT ;
 12
 13
 14
 15
OK


SCR #22
  0 ( L. L. Odette     6-2-83                          MVP-FORTH )
  1 ( Figure 1, Screen 2 of 2: INSERT and PERMUTE words )
  2 : INSERT    DUP 0=              ( Check to see if list is empty )
  3             IF 1+               ( Add new member if empty )
  4             ELSE INVERT 1-      ( Insert latest in list-last )
  5                  ONEOF +INVERT  ( Insert latest at end )
  6                        MYSELF 1+ ( Or recurse )
  7             THEN ;
  8
  9
 10 : PERMUTE    DUP IF 1- MYSELF INSERT THEN ;
 11
 12
 13
 14
 15
OK


 0 , ( Find a random permutation of the list on the stack ) OK
1 2 3 4 5 6 7 8 9 9 PERMUTE .S  ( List is originally in order )
4 1 8 2 5 3 6 9 7 9
OK
NONE .S    ( Again )
4 8 1 2 5 3 6 9 7 9
OK
NONE .S    ( Again )
4 8 2 5 3 1 6 9 7 9
OK
NONE .S    ( One more time )
4 8 2 5 3 6 1 9 7 9
OK
```

Figure 1.

Arbitrary permutation of a list using PERMUTE

The main word in the definition of permute in INSERT, which inserts an element X at a random place in a list of length N. *The list is* assumed to be on the stack in the form X, $X_N, \ldots X_1$, N (top of the stack to the right). PERMUTE reduces the list length by 1 and then calls itself. The return from the recursive call uses INSERT to rebuild the list by inserting the elements in random positions. PERMUTE expects a list of length N on the data stack in the form $X_N, X_{N-1}, \ldots X_1$, N. The definition and a description of the word NONE is given in the text.

```
SCR #34
  0 ( L. L. Odette     6-2-83                                MVP-FORTH )
  1 ( Figure 2 , Screen 1 of 2: 8 Queens Problem )
  2   VARIABLE BVAR                          ( Boolean Variable )
  3 : TRUE   DROP 1 BVAR ! ;                 ( Set Boolean true )
  4
  5 : ?ATTACKS   0 BVAR ! OVER               ( Init Boolean false )
  6              IF OVER 0
  7                DO OVER I - 2+ PICK       ( Get row placement )
  8                   OVER OVER =            ( In same row ? )
  9                   IF TRUE LEAVE          ( Yes - attack possible )
 10                   ELSE OVER 4 PICK - OVER I - =    ( Diagonal? )
 11                     IF TRUE LEAVE        ( Yes - attack possible )
 12                     ELSE OVER 4 PICK + OVER I + = ( Diagonal? )
 13                       IF TRUE LEAVE ELSE DROP THEN
 14                     THEN THEN
 15                LOOP THEN BVAR @ ;       ( Leave flag on stack )
OK


SCR #35
  0 ( L. L. Odette     6-2-83                                MVP-FORTH )
  1 ( Figure 2, Screen 2 of 2: 8 Queens Problem )
  2  : RANDOMN   DUP IF 1- ONEOF MYSELF 1+ THEN ;
  3
  4  : TRYQUEEN  DUP 2+ PICK RANDOMN ;  ( Choose a row placement)
  5  : ADDQUEEN  SWAP 1+ ;               ( Add row placement )
  6  : ?LASTQUEEN DUP 2+ PICK OVER <     ( Check column count )
  7               ?TERMINAL IF DROP 1 THEN ; ( Check keyboard too )
  8  : PRINTSOL  CR DUP 4 .R ." QUEENS" ( Print row placement )
  9              0 DO 1+ 4 .R LOOP DROP 4 SPACES ;
 10
 11  : QUEENSOLN   1- 0
 12                BEGIN  TRYQUEEN
 13                  ?ATTACKS IF NONE ELSE ADDQUEEN THEN
 14                  ?LASTQUEEN UNTIL
 15                PRINTSOL ;
OK

0 , 8 QUEENSOLN  ( Solves for 8 Queens - 0 indicates dictionary end )
   8 QUEENS   8   3   1   6   2   5   7   4    OK
NONE   ( Backtrack to find a different solution )
   8 QUEENS   2   8   6   1   3   5   7   4    OK
NONE   ( More )
   8 QUEENS   6   1   5   2   8   3   7   4    OK
NONE   ( More )
   8 QUEENS   3   6   2   5   8   1   7   4    OK
NONE   ( Even more )
   8 QUEENS   6   3   7   2   8   5   1   4    OK
NEW    ( Reclaim dictionary ) OK
0 , 12 QUEENSOLN  ( Solves for 12 Queens ! )
  12 QUEENS   9   4   6  11   2   7   1   3   5   8  10  12    OK
NONE   ( More )
  12 QUEENS   7   4   6  11   9   1   3   5   2   8  10  12    OK
```

Figure 2.

Solution of the N Queens problem using QUEENSOLN

QUEENSOLN expects the board size N on the stack and returns a list of N row placements for N queens in the columns from 1 to N. Execution of NONE will result in backtracking and the computation of a new solution. The word NEW is used to reclaim the dictionary space used by ONEOF.

(Contined from page 46)

ministic primitives provides significant advantages over explicit programming of backtracking. The advantages of nondeterminism are also apparent for implementing various randomizing algorithms, such as the random permutation program of Figure 1.

Purists will note that there is no such thing as random choice operating in any of the programs we have discussed. This is because we've used a pseudo random number generator in the definition of ONEOF to determine where to transfer control. In fact, each time ONEOF is executed, transfer of control is determined entirely by the state of the pseudo random number generator (the contents of the variable RND; see screen 30 in Listing One).

Notwithstanding this technicality, ONEOF and NONE work quite well. One of the more interesting effects is that QUEENSOLN takes different times to run on successive executions. Observations of this sort are reasons why probabilistic algorithms are an intriguing area for research. For example, keeping in mind that the randomness is not in the input data but in the way the program executes, we might ask the following question: Given a deterministic algorithm that takes a certain time to run to completion, can a probabilistic algorithm be created that has a faster mean time to solution?

Acknowledgements

I would like to thank Don Ruffle for his insightful comments and suggestions for improving the manuscript.

Notes

[1] P. Henderson, 1980, *Functional Programming, Application and Implementation,* London: Prentice Hall International, Inc.

[2] M. O. Rabin, 1977, Complexity of Computations, *CACM* 20:625; and 1979, Probabilistic Algorithms in Finite Fields, M.I.T. Lab for Comp. Sci. Memo No. 213.

[3] M. O. Rabin, 1976, Probabilistic Algorithms, in *Algorithms and Complexity, New Directions and Recent Trends,* ed. J. F. Traub, New York: Academic Press, pp. 21-39.

[4] J. LeVan, 1980, Recursion – The Eight Queens Problem, *Forth Dimensions* II:6.

[5] Henderson.

[6] LeVan.

(Listing begins on page 52)

4 183 287 391 + - *
Learn Forth quickly I will.

– Cartoon by Mary M. Milewski

Nondeterministic Control - Listing

(Text begins on page 44)

```
SCR #30
  0 ( L. L. Odette     6-2-83                              MVP-FORTH )
  1
  2    VARIABLE RND    HERE RND !          ( Random Number Words )
  3
  4 : RANDOM    RND @ 31421 * 6927 + DUP RND ! ;
  5 : CHOOSE    RANDOM U* SWAP DROP ;
  6
  7 : MYSELF    LATEST PFA CFA , ;    ( Control word for recursion )
  8             IMMEDIATE
  9
 10 : NEW       -2 ALLOT HERE @       ( Dictionary Reclaimation )
 11             IF   BEGIN HERE @ 2+ NEGATE ALLOT
 12                        HERE @ 0=
 13                  UNTIL
 14             THEN ;
 15
OK

SCR #31
  0 ( L. L. Odette     6-2-83                               MVP-FORTH )
  1
  2 ( Words used to save and restore the data stack )
  3
  4 : DATA>    SP@ DUP S0 SWAP - DUP  ( Get length in bytes )
  5            ROT HERE ROT CMOVE     ( Move to dictionary )
  6            DUP ALLOT , ;          ( Adjust dictionary pointer )
  7
  8 : >DATA    -2 ALLOT  S0 DUP >R    ( Save initial stack address )
  9            HERE @ - SP0 ! SP!     ( Reset stack pointer )
 10            R> SP0 !               ( Reset initial stack address )
 11            SP@ HERE @ DUP         ( Get address for move )
 12            NEGATE ALLOT           ( First adjust disctionary )
 13            HERE ROT ROT CMOVE ;   ( Restore data stack )
 14
 15
OK

SCR #32
  0 ( L. L. Odette     6-2-83                               MVP-FORTH )
  1
  2 ( Words used to save and restore the return stack )
  3
  4 : RETURN>  RP@ 2+ R0 @ OVER -     ( Calculate length in bytes )
  5            DUP ROT HERE ROT CMOVE ( Ret-stack to dictionary )
  6            DUP ALLOT , ;          ( Adjust dictionary pointer )
  7
  8 : >RETURN  R> -2 ALLOT            ( Save top of return stack )
  9            R0 @ DUP HERE @ -      ( Compute stack pointer )
 10            R0 ! RP! R0 !          ( Reset stack pointer )
 11            HERE @ DUP
 12            NEGATE ALLOT           ( Adjust dictionary pointer )
 13            HERE RP@ ROT CMOVE     ( Restore return stack )
 14            >R ;                   ( Restore old top of stack )
 15
OK

SCR #33
  0 ( L. L. Odette     6-2-83                               MVP-FORTH )
  1 ( ONEOF and NONE control words )
  2
  3 : <ONEOF>  R> DUP 4 + >R          ( Increment return pointer )
  4            DUP 2 CHOOSE           ( Toss coin )
  5            IF 2+ >R               ( Save pointer to first )
  6            ELSE >R 2+ THEN        ( or second word )
  7            RETURN>                ( Copy return stack )
  8            R> DROP >R             ( Juggle pointers )
  9            DATA>                  ( Copy data stack )
 10            R> @ EXECUTE ;         ( Execute one not saved )
 11 : ONEOF    COMPILE <ONEOF> ; IMMEDIATE
 12
 13 : NONE     HERE 2- @                ( Test data length )
 14            IF >DATA >RETURN R> @ EXECUTE ( Restore and continue)
 15            ELSE CR ." ***End of data*** " THEN ;
OK
```

End Listing

GO in Forth

by C. H. Ting

C. H. Ting, Offete Enterprises, 1306 South B Street, San Mateo, California 94402.

Copyright © 1982, 1983 by C. H. Ting. All rights reserved.

Permission is granted for personal, non-commerical use only. Any use for profit or other commercial gain without written permission of the author is prohibited.

Dr. Jonathan K. Millen published a very interesting article on "Programming the Game of Go" in the April issue of *Byte* magazine in 1981. It was a milestone in the computerization of this ancient Chinese board game. There were two major contributions in this work. First was the very simple algorithm for analyzing the progress of the game by counting the liberties of connected groups of stones. The liberty count became the priority scale that could be used to decide the next move by the computer. Second was the demonstration that, using this approach, a very small microcomputer could be programmed to play the Go game, which had always been considered to be too complicated for small computers.

This simple algorithm cannot cope very satisfactorily with all the intricacies of Go; however, it does provide a very firm foundation for future improvements and embellishments. The fact that Millen's program is capable of playing legitimate games, with only minor exceptions, demonstrates well that his algorithm captured the most essential features of the game. It forms a backbone of advanced Go programs in which other features can be added readily to deal with tactics and strategy.

I took Dr. Millen's article as a challenge to a programmer's skill to implement it on a small computer. His implementation was on a KIM-1 with 1 Kbyte of memory. The response time for a move was about one second. These features are hard to match. It would be interesting to see if it could be put on other computers, preferably in a high-level language to make the program transportable. Forth was chosen because of its compactness and efficiency in both programming and runtime. This game was programmed and used as both a practical example and a teaching example of the Forth language in late 1981, when I was back in Taiwan trying to preach the gospel of Forth. It was very effective because the difficulty in this game was well understood over there. A working Go game, in spite of its shortcomings, was impressive.

The Algorithm

The algorithm to play the game of Go as published by Dr. Millen is shown in Figure 1 (page 55). For a more detailed description of the mechanism and rationale in this program, the interested reader should consult the original article. However, the structured English he used to express the algorithm is not difficult to comprehend.

Implementation

The program was first developed on an LSI-11 microcomputer with Poly-Forth as the operating system. The source code spans 8 Kbytes, though loosely packed. The compiled code, including 362 bytes to hold the complete 19-by-19 board, occupied about 3 Kbytes. The response time for a move is about seven seconds.

Later, after I had finished a Forth-79 system for the Apple II and had succeeded in putting Forth in PROMs (replacing the Applesoft BASIC chip set), I tried to transfer the Go game into the Apple. Since the source code is less than 8 Kbytes, it was burned into four 2716's and moved into the Apple on an Apple ROM card. The Forth system in the Apple compiles the source code in ROM to object code stored in RAM and executes the Go game. The interesting thing about this scheme is that I can run a number of games with the same Apple, with all the games on ROM cards in source form.

Transportability

To make sure that the program can be transported to other microcomputers, it is important to avoid memory-mapped graphic representation of the Go board. Standard terminal I/O commands are used to draw the board with stones on the CRT screen. The only terminal-dependent commands are HOME and EOL (HOME returns the cursor to the upper left corner of the screen and EOL erases a line of characters). These two commands manage a stationary board on the screen and facilitate the key-in of the player's moves. They are not necessary if one does not mind the scrolling of the game board.

Although Forth-79 standard was used to present the program, it can be readily modified to fig-Forth or other Forth systems. In fact, I have seen the program running on a couple of CP/M fig-Forth systems, some of which I implemented for local computer manufacturers in Taipei for demonstration purposes.

The Board

Let us turn to the source screens in Listing One (page 63) to see how the algorithm is implemented.

The first two lines in screen 160 declare some constants and variables. BLACK is the code of black stones, having a constant value of 1. Similarly, WHITE has a constant value of 2. COLOR is a variable used to transfer the code of the stone groups currently under processing. LIBERTY is the liberty count (discussed below) accumulated for a group of connected stones; this is the primary indicator of the priority for attention to a given group of stones. MAP is an array of 362 bytes, storing the current board configuration. Each location on the 19-by-19 grid is designated as a byte in this array. A zero in a byte indicates an empty grid location, a 1 indicates a black stone, and a 2 indicates a white stone.

Skipping the rest of screen 160, let us look at the source code in screen 161, because it deals with the display of the game board. HOME moves the cursor to the upper left corner of the screen, preparing to write the new board configuration over the old board. It is desirable to keep an image of the old board on the screen. HOME is written in assembly code, calling the Apple Monitor routine to do the trick.

BK, WT, and CROSS respectively print a black stone, white stone, or grid cross point on the screen. A black stone is the "@" character. A white stone is the capital "O." The grid crossing was first represented by the "+" sign, but later replaced by the period because the "+" signs are too heavy and tend to obscure the stones on the CRT screen.

LIMITS places two numbers, 361 and 0, *on the stack because* they are often used to designate the loop limits in the analysis of board positions.

INDEX prints a line of indices on the top of the board to designate the columns of the board. BOARD prints the complete board on the screen, with indices on top and left. It takes the contents of the array MAP and prints the appropriate stones on the board. The MAP array is

initialized by CLRMAP, which erases the array to zeros.

PLACE stores a code into one of the array elements. The top of stack has the board position; the next item is the code to be stored in that location. KSTONE removes a stone from the board by storing a zero in the corresponding array element. These two commands change the board configuration. PUT is a utility to put a contiguous block of stones on the board for testing purposes. The color of the stones is specified by the contents of COLOR.

Base 19 Numbering System

GOBASE stores 19 into the user variable BASE, specifying that all subsequent number conversions are to be done in base 19. This is the most appropriate number base for the Go game, since it is played on a 19-by-19 board. The advantage in using base 19 is that all rows and columns can be designated with one character from 0 to 9 and A to J. When typing in a move, the player can type one number – e.g., 35, AF, 9J, etc. – to represent a board location. This number is the exact offset to pick up the proper array element in MAP, without any complicated conversions. The freedom in choosing the best number base for a specific application is a luxury only a Forth programmer can enjoy.

A very good example of how to use GOBASE is the command HANDICAP, which places nine handicap stones on the board to give the computer a fighting chance to compete with a human player. After switching the base to 19 via GOBASE, the board positions can be conveniently specified as 33, 3A, 3F, etc., and PLACEd on the board in a short loop. After HANDICAP, the base is returned to decimal for normal processing.

Liberty Counts

The most important procedure in Millen's algorithm is that which scans a group of connected stones of the same color, and counts the liberties, or the

Listing 1: *Structured English specifications of COUNT module to find and count the liberties of a connected group containing a stone at point "x" of color "color." COUNT calls itself recursively, saving x on the push-down stack during each call.*

```
COUNT(x, color):
  IF x is not off the edge
  THEN
    IF there is a stone at x AND
      it is the given color AND
      it is not marked
    THEN
        mark it
        CALL COUNT(NORTH(x), color)
        CALL COUNT(EAST(x), color)
        CALL COUNT(SOUTH(x), color)
        CALL COUNT(WEST(x), color)
    ELSE IF there is no stone at x
    THEN
        mark the point as a liberty
        increment the liberty count
    END
END
```

Listing 2: *Module specification for the main loop of the Go-playing program and two of its called modules.*

```
MAIN:
  place black handicap stones
  LOOP
    display the board
    get white's move from keyboard
    CALL WEFFECT for the effect of white's move
    CALL BEFFECT to obtain a tentative black move
    CALL PATS to check for a pattern match
    place black stone
END

WEFFECT:
  FOR each point x with a black stone DO
    CALL COUNT(x,black)
    IF the group has no liberties
    THEN remove its stones
    ELSE IF the group has at least one liberty
    THEN
      choose a liberty not on edge line
      IF the group has 1 or 2 liberties
      THEN CALL EVAL for the chosen liberty
    END
END

BEFFECT:
  FOR each point x with a white stone DO
    CALL COUNT(x,white)
    IF the group has exactly 1 liberty
    THEN
      designate it as the black move
      remove the white stones
      EXIT
    ELSE IF the group has 2 or more liberties
    THEN
      choose a liberty
      CALL EVAL for the chosen liberty
    END
END
```

Listing 3: *Module specifications for move evaluation, look-ahead, and pattern matching.*

```
EVAL(move,liberties):
  GLOBAL (best-move, best-liberties)
  IF liberties ≤ best-liberties AND
    LOOKAHEAD(move) ≥ 2
  THEN
      best-move = move
      best-liberties = liberties
  END

LOOKAHEAD(move):
  place black stone at move
  CALL COUNT(move,black)
  remove black stone
  RETURN count of liberties

PATS:
  FOR each white stone DO
    IF there is a pattern in the table
      centered on that white stone
    THEN
      get suggested black move y
      CALL EVAL(y,2)
      EXIT
  END
END
```

Figure 1.

From "Programming in the Game of Go" by Jonathan Millen, originally published in the April 1981 issue of *Byte* magazine. Used with the written permission of Jonathan K. Millen.

open grid positions, around this group. This procedure is implemented in screen 160.

Let us follow the order in which commands are defined in screen 160. ?OUT takes the number at the top of the stack and checks its range. If the number is between 0 and 360 inclusive, a false flag is returned; otherwise, a true is returned. It will be used to abort a counting procedure if the stone location is outside of the board.

?STONE merely returns the code of the stone whose location is given on the top of the stack. The top of stack number is used as an offset into the MAP array to pick up the code stored in the specified location.

NORTH, SOUTH, EAST, and WEST are the commands that convert the current stone location on top of the stack to the neighboring location in the direction indicated by the names of the commands. The returned location might be outside of the board, as NORTH and SOUTH would do if the given location is at the edge of the board. In the cases of EAST and WEST, there is the problem of horizontal wrap-around. If the current location is at the left edge, WEST will then return the number 1000, which is definitely outside of the board. EAST will do likewise at the right edge.

MARK sets the most significant bit of the code whose location is on the top of the stack. This command puts a mark on the locations that were scanned at least once so that the location will not be repeatedly scanned or repeatedly counted.

RECUR is the famous Fig-Forth command allowing compilation of recursive procedures. (Other versions of Forth may use other words, such as MYSELF or RECURSE.) It compiles the code field address of the procedure, which is still in the process of compilation, allowing it to call itself at runtime.

COUNTS walks through an entire group of connected stones and accumulates its liberties in the variable LIBERTY. The color of the stone group is specified in the variable COLOR. It walks by the left-turn rule of the maze theorem, examining the locations north, east, south, and west of the current location. If that neighboring location has an unmarked stone of the same color, it will jump into this location and call itself to continue the walking. It will increment the LIBERTY count if the neighboring location is empty and unmarked. It marks every location it examines so that locations will not be repeatedly scanned or counted. At the end of this recursive process, all the connected stones of the same color will have been scanned and all the liberties around this group will have been marked and counted.

The recursive procedure is naturally limited in its depth by the sizes of the stacks allocated by the system. In this implementation each level of call uses one cell of the return stack and one cell of the data stack. Since in most Forth systems the return stack has about 256 bytes, COUNTS can process a group of 100 connected stones. This is adequate for most games normally played. The system will crash if the stacks overflow.

Removing Stones

In screen 162, the important commands are DESIGNATE and REMOVE. DESIGNATE is to be used in the case that a group of white stones has only one liberty left. The computer will immediately kill this group by playing a black stone into the liberty. After the counting process, this liberty location is the only location having a code of 128, empty and marked. DESIGNATE simply scans the board, and upon finding this location, plays the killing move.

REMOVE erases an entire connected group of stones, whose color is designated in COLOR. REMOVE is very similar to COUNTS, using the same recursive technique to scan the connected group. It marks the member stones in the group by setting the seventh bit of the code as the kill mark, so that the stone scanned will be deleted at the end of the REMOVE process. KMARK performs this marking task.

After the counting or the removing process, many locations examined are tagged by the counting marks or the kill marks. To continue the processing of other stone groups, these marks must be removed to restore the board map to a clean state. This is accomplished by the command UNMARK. UNMARK scans through the whole board and resets all the bits in the codes according to the reset pattern given on the top of the stack. This way, it can be used to selectively reset either the counting bits, the kill bits, or both at the same time.

Lookahead

This Go program only has the ability to look ahead by one step, which is one of its many weaknesses. However, one-step lookahead does give it some similarity to an amateur player. Screen 163 contains the commands to examine the board configuration one step ahead.

The variables BEST-MOVE, BEST-LIBERTIES, BEST-COUNT, and !COLOR are temporary storage locations used to determine the best move in a given board configuration. The command LOOKAHEAD takes a board location given on the top of the stack, tentatively places a black stone in this location, and counts the liberties around the group of black stones connected to this stone. The liberty count is returned to the stack, and the black stone is removed from this location. A better move for the black would have a larger liberty count. The liberty count is then compared to that in the BEST-LIBERTIES to decide whether BEST-MOVE needs to be updated in the later command.

EVAL does the comparison and updating. It takes two values off the stack; the top value is the maximum liberty count allowed in the current configuration and the second is the move to be examined. EVAL calls LOOKAHEAD to get the liberty count of the proposed move. If the resulting liberty count is greater than the count in BEST-COUNT, and the maximum count given on the stack is not greater than the count in BEST-LIBERTIES, the proposed move should be a better choice than the one indicated in BEST-MOVE as the result of prior analysis. In this case the variables BEST-MOVE, BEST-COUNT, and BEST-LIBERTIES are updated. Otherwise, the best move as determined by prior analysis remains intact.

Going through this evaluation process for all the black groups, the computer will be able to pick up the weakest black group and find the best move to maximize its liberty count; hopefully this will save the black group from being captured by the white stones. In attacking the weakest white group, it also chooses the most secured move so that the attacking stone will not be threatened easily.

Pattern Recognition

The following two screens, 164 and 165, contain commands dealing with some elementary pattern recognition, designed to identify some favorable positions to be considered in the evaluation process. Dr. Millen identified seven patterns in which a designated black move usually improves on the overall black configuration. These seven patterns, including some permutations and reflections, are reduced to twelve 16-bit patterns stored in an array named PATTERN. The coding scheme is illustrated in Figure 2 (page 59). An empty location will be examined by coding the stone distributions in the neighboring 3-by-5 regions to its immediate north, east, south, and west. If any of the four codes matches with one of the twelve patterns, this empty location is assigned a liberty count value of 2, a rather high priority.

?RANGE takes a board location off the stack and returns the code stored in the location. The difference from ?STONE is that if the location is outside of the board, ?RANGE will return a 1, the code for black stone. This is the "ghost stone" asserted by Dr. Millen. The entire board can be considered as being surrounded by black stones to take into account some favorable conditions near the edges of the board.

Given a particular board location,

?N, ?E, ?S, and ?W return the code of its northern, eastern, southern, and western neighbor, respectively. They use ?RANGE to find the codes. If the neighbor is off the board, the code of a black stone will be returned.

+4* is a strange command that generates a 16-bit code according to the rules for the codes being constructed in PATTERN. It takes eight items off the stack and packs the least significant two bits in each item into a 16-bit code, to be left on the stack. These items are the codes of stones immediately around a location, and their order on the stack is according to the sequence specified in Figure 1.

PNORTH is given a specific location *on the stack* and looks at its northern neighboring location. If this neighbor is empty, PNORTH will walk through its eight neighboring locations, push their codes of stones on the stack, and call +4* to pack these eight codes into a single 16-bit pattern code. However, if the northern neighbor is not empty, PNORTH will exit immediately because this neighboring location is not playable and does not need pattern matching.

PEAST, PSOUTH, and PWEST behave similarly. It seems rather clumsy that four similar words have to be defined to do very similar tasks. Some better tools are needed here to clean up the codes.

MATCH is the command that does the pattern matching. It takes a coded pattern off the stack and compares it with the patterns stored in PATTERN. If the coded pattern matches with one of the stored patterns, a true flag is left on the stack; otherwise, a false flag is left. The big command PATS scans the whole board looking for white stones. Finding a white stone, PATS will use PNORTH, PEAST, PSOUTH, and PWEST to examine its surroundings to see if a stored pattern could be identified. It will abort the loop at the first sight of a matched pattern, assign a priority of 2 to this pattern, and call EVAL to do an evaluation. Since PATS is the last pass in the evaluation process, its finding is preferred over other evaluations of the same priority.

In matching a coded pattern with the stored patterns, I initially used the comparison command = to match the patterns. This turned out to be a very restrictive operation because among the eight neighbors there are two or three considered to be crucial – the others can be ignored. FIX takes the coded pattern and ANDs it with the stored pattern to eliminate the non-crucial bits in the coded pattern. The resulting modified pattern is then compared with the stored pattern to determine a match. However, adding FIX to MATCH tends to make the comparison too liberal. It is probably necessary to define a set of masks to be used together with the patterns in order to be able to precisely identify the desired patterns.

Analysis

Screen 166 hosts the two major analysis routines: WEFFECT, which analyzes the effects of the white stones, and BEFFECT, which analyzes the effects of the black stones. The variable ?STOP and the command STOP (which clears ?STOP) will be used later in the main loop to bypass the pattern recognition process if the computer makes a capture. It is rather cumbersome to transmit this stop flag on the stack. A variable is needed to do the job.

UNMARKS initializes the move analysis routines: WEFFECT, which analyzes bits in the MAP array and storing the color of the stones to be processed, given on the stack, into COLOR. EXAMINE takes a board location and calls COUNTS to count the liberties of the group of stones connected to this location; EXAMINE then returns the liberty count on the stack.

WCHO and BCHO are two commands that scan the board for empty but

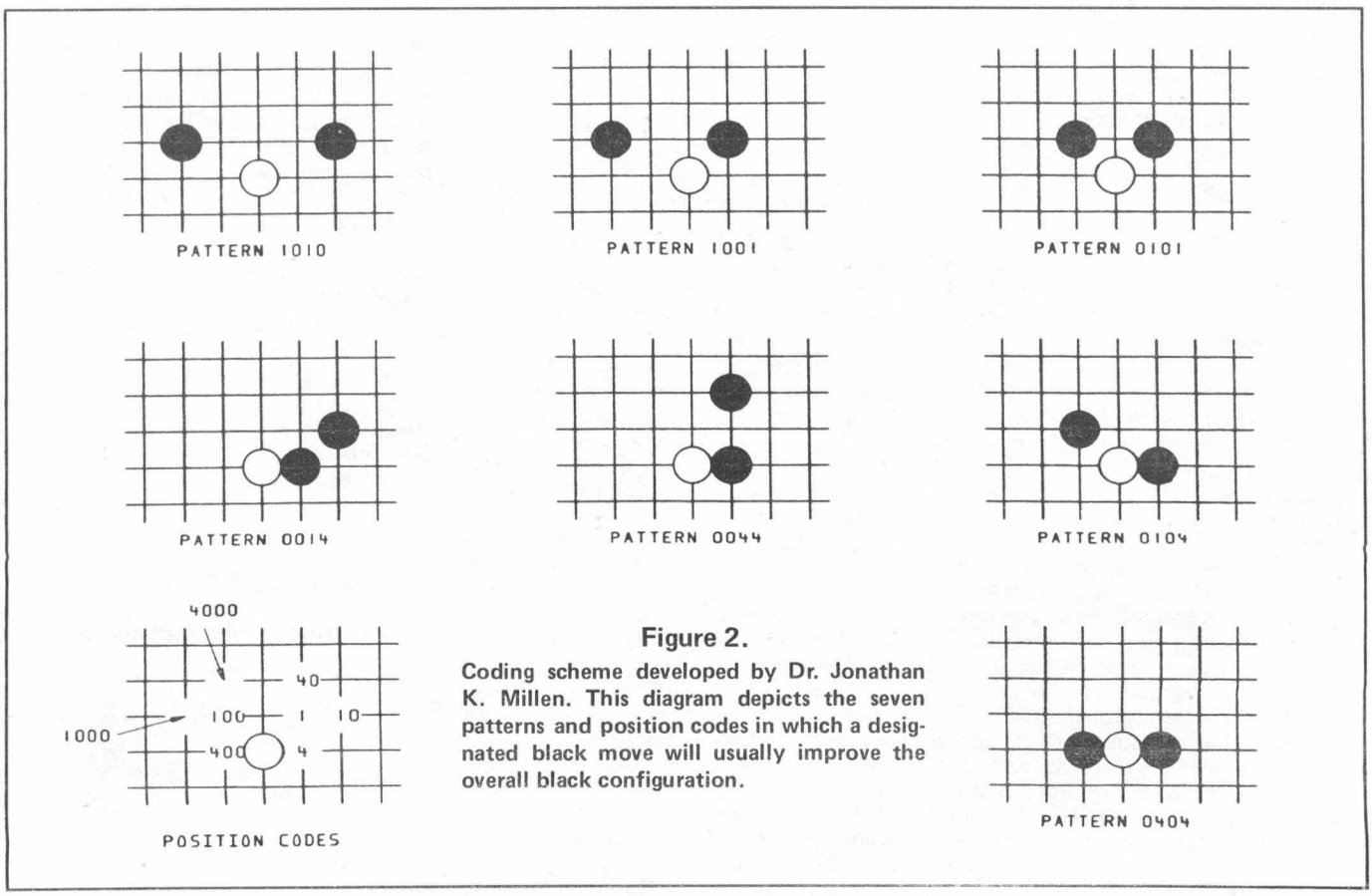


Figure 2.

Coding scheme developed by Dr. Jonathan K. Millen. This diagram depicts the seven patterns and position codes in which a designated black move will usually improve the overall black configuration.

marked locations and push these locations on the stack. These locations are candidates for the next black move, and are to be analyzed using the lookahead technique. A zero is pushed on the stack first as a floor for the location values. These locations will be picked up by CHO to do the actual analysis. CHO scans the locations on the stack and analyzes their priorities. When it finds a zero on the stack, it terminates the analysis. The choice of the best move is stored in BEST-MOVE. WCHO differs from BCHO in that it does not scan the edges of the board, because it is futile for black to play on the board edges to avoid capturing.

?CHO is used to determine whether the liberty count of the current stone group is greater than the value stored in BEST-LIBERTIES. If the liberty count is greater, there is no need to do the move analysis because this move is of a lower priority.

WEFFECT scans all the black stone groups. If a black group has a liberty count of zero, it is completely surrounded by white stones and must be removed from the board. If the liberty count is less than 3 and equal or less than BEST-LIBERTIES, an analysis is performed. If a more pressing move is found, the contents of BEST-MOVE, BEST-COUNT, and BEST-LIBERTIES are updated.

BEFFECT scans all the white stone groups. If a white group has a liberty count of 1, the computer will capture this group by playing into the last liberty. The STOP flag will be cleared at this point. Otherwise, BEFFECT will pick the best attacking move.

Terminal Input

?MOVE in screen 167 is the command that accepts a number from the keyboard as the next move by the human player. It prints a prompting message, clears the rest of the line, and waits for an input string from the keyboard, terminated by a carriage return. It checks the range of the number and also whether the board location is occupied. It will loop until a valid number is entered. Since it uses the standard NUMBER command to do the number conversion, it will abort to the interpreter if an invalid number is entered. This turns out to be a convenient way of stopping the Go game for whatever purpose. The aborted game can be resumed at the point of interruption by the command RESUME.

EOL is the command to clear the rest of the current line. It calls an Apple monitor routine to do the job. It is not absolutely necessary if the player is aware of the way the screen display is managed.

It is important to use numbers in the base 19 format. This way a location is selected by a two-digit number from 00 to JJ, without spaces between the two digits. The columns and the rows of the board are numbered accordingly to facilitate the input process.

The Main Game Loop

The command for playing the Go game is RESUME, which is an infinite loop. In this loop, first the current board and stones are printed out on the CRT screen, the player is asked to input his move, and the computer then goes through the WEFFECT, BEFFECT, and PATS routines to decide its best move. The computer does not know when to stop. It is the human player who usually gets worn out and quits.

.BEST is used to print out the variables stored in BEST-MOVE, BEST-COUNT, and BEST-LIBERTIES. Printing out this information is interesting, as it shows how the computer arrives at its final moves. Since the computer takes about eight seconds to decide its move, printing the intermediate results assures the player that the computer is not idling.

MAIN is the initial entry point of the game. It clears the board, places the nine handicap stones on board, sets the base to 19, and drops into the RESUME loop to play the game. If the game is interrupted by intentionally or unintentionally entering an invalid number, a new game can be started by MAIN or the current game can be continued by RESUME.

Concluding Remarks

The resulting program, compiled into Forth code, occupies about 3 Kbytes of memory. The response time for each move is about eight seconds. These statistics cannot match those of Dr. Millen's implementation on KIM-1 at 1 Kbyte and one second response time. Inefficiency in the high-level codes and poor understandings of the original algorithm are among the most obvious reasons for the decrease in performance. I don't think much can be done in the area of high-level language, since Forth is just about the best high-level language for this type of application. To really make it run faster, many of the routines will have to be coded in assembly. I wonder if anybody will do it, except Dr. Millen.

There are many possible ways of improving the performance of this Forth program. Restricting the scope of searching and analysis will greatly speed up the execution. I tried to limit the searching to two 5-by-5 areas around the black stone last played and the white stone just played. This shortcut speeds up the response time to less than two seconds, without significantly weakening the performance of the computer part of the game. Recording the results of a whole pass of analysis and using them for the next pass will reduce much of the redundant work performed in each pass. The results of analysis need only minor updating and the best move can be quickly selected in the next pass.

There are also some more fundamental problems not properly addressed in the original program, like ko fights, the ladder configuration, Joseki's, etc. These problems can be solved by adding more complicated analysis routines to the framework. The most serious problem is probably that of recognizing two eyes in a connected group of stones and of generating two eyes to build a secured group, immune from being captured. The counting algorithm is not capable of doing this job. To accommodate this capability, a radically different algorithm is needed to do much more sophisticated pattern analysis. A full-scale connectivity analysis could be the best solution. This route is currently being explored.

There will be no end to the lookahead analysis. Professional players look ahead ten or more steps, which is very difficult to simulate by computer. Probably a two- or three-step lookahead in combination with a more extensive pattern analysis process will be sufficient. An important area is the opening games or Joseki's. Large collections of Joseki's compiled into data bases can be useful in guiding the initial games in the four corners on the board. Not much intelligence is needed besides searching the data base for the particular opening. The same is true for the game endings. However, it will then be necessary that the computer have disk storage to hold the data bases.

Computerizing the game of Go is very interesting and challenging. I heartily congratulate Dr. Millen for opening up this field with his excellent work, and hope that many people will also contribute their expertise to a fuller implementation of this game on the microcomputer. Hopefully, we will have the computer Go game reach the level of sophistication of the chess games now available commercially.

DDJ

(Listing begins on page 63)

GO - Listing (Text begins on page 54)

```
160 LIST

 0 ( STONES, CHT,10-1-82)
 1 VARIABLE COLOR VARIABLE LIBERTY 1 CONSTANT BLACK
 2 2 CONSTANT WHITE                 VARIABLE MAP 360 ALLOT
 3 : ?OUT DUP 0< OVER 360 > OR ;    19 CONSTANT 19
 4 : ?STONE ( N --- CODE ) MAP + C@ ;
 5 : NORTH ( N1 --- N2 )  19 - ;    : SOUTH 19 + ;
 6 : EAST  DUP 19 MOD 18 =          IF DROP 1000 ELSE 1+ THEN ;
 7 : WEST  DUP 19 MOD IF 1-         ELSE DROP 1000 THEN ;
 8 ( COUNTS, 10-1-82 )
 9 : MARK MAP + DUP C@ 128 OR SWAP C! ;
10 : RECUR LATEST PFA CFA , ; IMMEDIATE
11 : COUNTS   ?OUT IF DROP EXIT THEN   DUP ?STONE COLOR @ = IF
12            DUP MARK   DUP NORTH RECUR   DUP EAST RECUR
13            DUP SOUTH RECUR   WEST RECUR
14         ELSE DUP ?STONE 0= IF MARK 1 LIBERTY +!
15            ELSE DROP THEN   THEN   ;


161 LIST

 0 ( DISPLAY, CHT, 10-1-82 )  HEX  : BK ."  @" ; : WT ."  O" ;
 1 CODE HOME XSAVE STX, 24 STY, 25 STY, FC22 JSR, XSAVE LDX,
 2          NEXT JMP, END-CODE     DECIMAL
 3 : LIMITS 361 0 ;                : CROSS ."  ." ;
 4 : IND 2 SPACES 19 0 DO I 2 .R LOOP ;
 5 : BOARD   HOME IND  LIMITS DO
 6           I 19 /MOD SWAP 0=     IF 2 .R ELSE DROP THEN
 7           I ?STONE 3 AND DUP 0= IF CROSS DROP
 8           ELSE BLACK = IF BK    ELSE WT THEN THEN LOOP ;
 9 : PLACE ( CODE N --- ) MAP + C! ;
10 : GOBASE 19 BASE !  ;           GOBASE
11 : HANDICAP 33 39 3F 93 99 9F F3 F9 FF   9 0 DO
12         BLACK SWAP PLACE LOOP ;                          DECIMAL
13 : CLRMAP   MAP 362 ERASE ;
14 : KSTONE ( N --- )   MAP + 0 SWAP C! ;
15 : PUT SWAP DO COLOR @ MAP I + C! LOOP ;


162 LIST

 0 ( REMOVE, CHT, 10-22-81)
 1 : KMARK   MAP + DUP C@ 64 OR SWAP C! ;
 2 : REMOVE ( N --- )
 3       ?OUT   IF DROP EXIT THEN
 4       DUP ?STONE 67 AND   COLOR @ = IF
 5          DUP KMARK   DUP NORTH RECUR   DUP EAST RECUR
 6     DUP SOUTH RECUR DUP WEST RECUR KSTONE ELSE DROP THEN ;
 7
 8 ( DESIGNATE, UNMARK, CHT, 10-1-82)
 9
10 : UNMARK ( PATTERN --- )   MAP 361 OVER + SWAP DO
11       I C@ OVER =   IF I C@ 3 AND I C! THEN   LOOP DROP ;
12 : DESIGNATE   LIMITS DO   I ?STONE 128 =
13       IF BLACK I PLACE LEAVE THEN   LOOP ;
14
15


163 LIST

 0 ( LOOKAHEAD, CHT, 10-1-82)
 1 VARIABLE BEST-MOVE   VARIABLE BEST-LIBERTIES
 2 VARIABLE BEST-COUNT      VARIABLE !COLOR
 3 : LOOKAHEAD ( MOVE --- LIBERTY )
 4       BLACK OVER PLACE   COLOR @ !COLOR !   BLACK COLOR !
 5       0 LIBERTY !   128 UNMARK   129 UNMARK
 6       DUP COUNTS   KSTONE   !COLOR @ COLOR !   LIBERTY @ ;
 7
 8 ( EVAL, CHT,10-1-82)
 9
10 : EVAL ( MOVE LIBERTY --- )
11       OVER LOOKAHEAD   >R I   BEST-COUNT @ 1 MAX >
12       OVER BEST-LIBERTIES @ > 0= AND
13       IF BEST-LIBERTIES ! BEST-MOVE !   R> BEST-COUNT !
14       ELSE 2DROP R> DROP THEN ;
15
```

(Continued on page 65)

GO - Listing

(Listing continued, text begins on page 54)

```
164 LIST

 0 ( PATS, MATCH, CHT, 10-1-82)   HEX
 1 VARIABLE PATTERN 14 , 44 , 1400 , 4400 , 1001 , 0110 , 404 ,
 2       401 , 101 , 104 , 1010 ,   1010 PATTERN !   DECIMAL
 3 : ?RANGE ( N --- CODE )   ?OUT IF
 4       DROP BLACK   ELSE ?STONE 3 AND THEN   ;
 5 : ?N NORTH ?RANGE ;              : ?E EAST ?RANGE ;
 6 : ?S SOUTH ?RANGE ;              : ?W WEST ?RANGE ;
 7 : 2* DUP + ;      : +4*   0 8 0  DO DUP + DUP + + LOOP ;
 8 : PNORTH ( MOVE --- PATTERN )   DUP ?N IF DROP 0 EXIT THEN
 9       >R  I NORTH ?E     I ?E     I NORTH EAST DUP ?E
10          SWAP ?N     I NORTH ?W     I
11       ?W     R> NORTH WEST DUP ?W     SWAP   ?N +4* ;
12 : PEAST   DUP ?E IF DROP 0 EXIT THEN
13      >R  I EAST ?S     I ?S     I EAST SOUTH ?S
14          I EAST SOUTH ?E     I EAST ?N     I ?N
15          I EAST NORTH ?N     R> EAST NORTH ?E +4* ;


165 LIST

 0 ( PATS, MATCH, CHT, 10-1-82)  : FIX DUP ROT AND ;
 1 : PSOUTH ( MOVE --- PATTERN ) DUP ?S IF DROP 0 EXIT THEN
 2       >R  I SOUTH ?E I ?E I SOUTH EAST ?E I SOUTH EAST ?S
 3       I SOUTH ?W I ?W I SOUTH WEST ?W R> SOUTH WEST ?S +4* ;
 4 : PWEST   DUP ?W IF DROP 0 EXIT THEN
 5      >R  I WEST ?S I ?S I WEST SOUTH ?S I WEST SOUTH ?W
 6          I WEST ?N I ?N I WEST NORTH ?N R> WEST NORTH ?W +4* ;
 7 : MATCH ( PATTERN--- F)   DUP IF   0 12 0 DO OVER PATTERN I 2*
 8       + @ FIX = IF 1+ LEAVE THEN   LOOP   SWAP DROP THEN ;
 9 : EVPAT    1 BEST-COUNT ! 2 EVAL ;
10 : PATS   LIMITS DO I ?STONE WHITE AND IF   I PNORTH MATCH
11       IF I NORTH EVPAT LEAVE    ELSE I PEAST  MATCH
12       IF I EAST EVPAT  LEAVE    ELSE I PSOUTH MATCH
13       IF I SOUTH EVPAT LEAVE    ELSE I PWEST  MATCH
14       IF I WEST EVPAT LEAVE   THEN THEN THEN THEN THEN LOOP ;
15


166 LIST

 0 ( WEFFECT, BEFFECT, CHT, 10-1-82)
 1 VARIABLE ?STOP   : STOP 0 ?STOP ! ;
 2 : UNMARKS ( COLOR---) 129 UNMARK 130 UNMARK COLOR ! ;
 3 : EXAMINE    128 UNMARK 0 LIBERTY !     COUNTS LIBERTY @ ;
 4 : WCHO 0 341 19 DO I 19 MOD ?DUP IF 18 - IF I ?STONE 128 =
 5       IF I THEN    THEN THEN LOOP ;
 6 : CHO  BEGIN ?DUP WHILE LIBERTY @ EVAL REPEAT ;
 7 : ?CHO BEST-LIBERTIES @ LIBERTY @ < 0= ;
 8 : BCHO 0 LIMITS DO I ?STONE 128 = IF I THEN LOOP ;
 9 : WEFFECT   BLACK UNMARKS   360 BEST-LIBERTIES !
10       LIMITS  DO   I ?STONE BLACK =   IF I EXAMINE
11       0= IF I REMOVE   ELSE LIBERTY @ 3 <
12       IF ?CHO IF WCHO CHO THEN THEN THEN THEN LOOP ;
13 : BEFFECT   WHITE UNMARKS   LIMITS DO   I ?STONE WHITE = IF
14       I EXAMINE 1 =   IF DESIGNATE I REMOVE LEAVE STOP
15       ELSE ?CHO IF BCHO CHO THEN THEN THEN   LOOP ;


167 LIST

 0 ( ?MOVE, CHT, 10-1-82)   HEX
 1 CODE EOL XSAVE STX, FC9C JSR, XSAVE LDX, NEXT JMP, END-CODE
 2 DECIMAL
 3 : ?MOVE BEGIN CR ." YOUR MOVE: "   EOL
 4         0 >IN ! TIB @ 19 EXPECT 32 WORD NUMBER DROP DUP . ?OUT
 5         IF ." RANGE? " DROP 0   ELSE DUP ?STONE 3 AND 0=
 6         IF WHITE SWAP PLACE 1   ELSE ." OCCUPIED." DROP 0  THEN
 7         THEN   UNTIL ;
 8 ( RESUME, MAIN, CHT, 10-1-82 )
 9 : .BEST  3 SPACES BEST-MOVE ? BEST-COUNT ? BEST-LIBERTIES ? ;
10 : RESUME   BEGIN   BOARD ?MOVE   1 ?STOP !   0 BEST-COUNT !
11         WEFFECT .BEST  BEFFECT .BEST    ?STOP @   IF PATS
12         BLACK BEST-MOVE @   .BEST  PLACE THEN   AGAIN ;
13 : MAIN  GOBASE CLRMAP HANDICAP RESUME ;
14 EXIT
15
```

End Listing

SEND and RCV:

A Forth Implementation of the XMODEM Protocol

by Robert Taylor

Robert Taylor III, 31 E. Morris Avenue, No. 10, Buffalo, New York 14214.

Copyright © 1982, 1983 by Robert Taylor III. All rights reserved. Not to be used for commerical purposes, but unlimited personal, non-commercial use is allowed.

In the dark ages after the IBM PC was delivered but before everyone jumped on the bandwagon to provide software for it, I needed to transfer files to and from CP/M systems. I knew that Ward Christensen's XMODEM program was the de facto standard for file transfer between CP/M systems, so all I had to do was write a program for the IBM PC that used the same protocol. Because I had just received a copy of PC/FORTH from Laboratory Microsystems and I wanted to learn Forth programming, I did the natural thing: I wrote SEND and RCV, Forth words that do XMODEM file transfers.

Before we get into the Forth words, perhaps a short introduction to the XMODEM protocol is in order. *Dr. Dobb's Journal* has published many articles over the years about various implementations of this protocol, so many of you are already familiar with it. For those of you who aren't, the protocol provides a method of transferring files and making sure that each piece of the file is transferred without error.

The XMODEM Protocol

Before anything is transferred, the sender and the receiver must get in synch with one another. To do this, the sender waits for the ASCII character NAK (Negative AcKnowledge, or decimal 21) to be sent by the receiver. When the sender gets the first NAK, it starts transferring the file one piece at a time.

The file is transferred in 128-byte chunks called "sectors." The sender precedes each sector with a 3-byte sequence called a "sector header." The sector header consists of the ASCII character SOH (Start Of Header, or decimal 1), followed by the sector number as an ASCII character (0-255), followed by the logical complement of the sector number (255-0). After the sector header is sent, first the sector and then a single byte called the "checksum" are sent. The checksum is the sum of every byte in the sector, modulo 256, so it will be a number from 0 to 255. When the sector header, the sector, and the checksum are sent, the sender waits for a response from the receiver.

The receiver receives the 3-byte sector header, the 128-byte sector, and the 1-byte checksum. It makes sure that the sector header is correctly formed. It saves the 128-byte sector into a file buffer and computes a checksum by adding all the bytes together, modulo 256. Then it compares its computed checksum with the checksum byte it received from the sender. If any characters have been received differently than they were sent, the two checksums will not have the same value, and the receiver knows that the sector is in error. If the receiver decides that no errors have occurred, it sends the ASCII character ACK (ACKnowledge, or decimal 6) to the sender. If the receiver finds an error, it responds with the ASCII character NAK.

The sender, having sent a sector, awaits the receiver's response. If the receiver responds with a NAK, an error has occurred, and the sender retransmits the same sector. If the receiver responds with an ACK, the sector arrived safely, so the sender sends the next sector in the file. When the sender runs out of sectors to send, it transmits the ASCII character EOT (End Of Transmission, or decimal 4) and waits for the receiver to transmit an ACK, signaling all's right at the receiving end.

If the line between the sender and the receiver is so noisy that nothing can be transmitted without errors, the sender will eventually give up. Also, if the sender receives the ASCII character CAN (CANcel, or decimal 24) when it is waiting for an ACK or a NAK, it will stop trying to send.

Using SEND and RCV

To use SEND or RCV, first load them into Forth. Since PARSE is a CODE word, the assembler must be already loaded. Once they are loaded into Forth, type

```
SEND filename.ext
```

to start the transmission of the PCDOS file specified. Similarly, type

```
RCV filename.ext
```

to start receiving the PCDOS file specified. As each sector is sent or received, its number is displayed on the screen so you can follow its progress. You can manually terminate the transfer at any point by hitting Ctrl-X at the keyboard. When the whole file is sent, RCV will display

```
EOT received
```

while SEND displays

```
EOT sent – ack'd
```

indicating that the sender has sent the required End of Transmission character and the receiver has received and acknowledged it.

Since these routines may be operating at high baud rates (up to 9600 baud), the receiver can't take the time to print long error messages. Thus, "H–" indicates a header error, "S–" a sector error, "C–" a checksum error, and "T–" a timeout error. (A timeout error is when no data have arrived within a certain period of time.) After an error occurs, "NAK sent" indicates that the receiver has sent a NAK to the sender. The NAK tells the sender to retransmit the sector.

Implementation of SEND and RCV

SEND and RCV are simply the Forth words that implement the actions of the sender and receiver. SEND first calls SEND-SETUP to find the file that will be transferred, to open it, and to report the file's size. WAIT-ACK waits for the receiver to send a NAK (or an ACK) to start the transfer. If WAIT-ACK receives a CAN, it closes the opened file and stops. Otherwise, SEND enters a loop that:

(1) Checks the keyboard for a Ctrl-X, which would cancel the transfer.

(2) Calls READ-SECT to get another sector from the file. If READ-SECT gets an end-of-file error, END-SEND is invoked to end the transfer normally.

(3) Calls SEND-HDR, SEND-SECT, and SEND-CKSUM to send the sector header, the sector, and the checksum, respectively.

(4) Waits for the receiver to acknowledge the transfer. If an ACK is received, the variable SSEC is incremented, so the next sector will be sent. If a NAK is received, SSEC isn't incremented, so the same sector will be sent again. In either case, the loop starts over from the beginning.

RCV works in a similar manner. RCV-SETUP makes sure that the file to

be received is properly created. If the file already exists, you are asked if you want to overwrite it. Then RCV sends the first NAK to get the ball rolling. Now RCV goes into a loop that:

(1) Checks the keyboard for a Ctrl-X, which would cancel the transfer.
(2) Waits for an ASCII SOH, which indicates the start of a sector header.
(3) Calls RCV-HDR to receive the header and check it for validity.
(4) Calls RCV-SECT to receive the sector and compute the checksum.
(5) Calls RCV-CKSUM to receive the checksum and compare it to the checksum computed by RCV-SECT.
(6) Calls SAVE-SECT to save the sector in the file on disk.
(7) Sends an ACK to the sender, and starts the loop over.

If RCV finds an error, it calls RCVERR, which waits for the sender to stop sending and sends a NAK. The sender then retransmits the sector, and RCV restarts the loop.

Conclusion

SEND and RCV provide a mechanism for the Forth programmer to transfer files to and from CP/M systems that have XMODEM and to and from other PCDOS systems that have a program that implements the XMODEM protocol. They were written for Laboratory Microsystem's PC/FORTH, which runs under PCDOS, but they are easily adaptable to other Forths running on other machines.

Since SEND and RCV are intended to transfer files, some file handler words must be added to most Forth implementations. One possible set of file handler words is presented in screens 2-6. These words are included in PC/FORTH, although I've made a few modifications and additions. To adapt SEND and RCV to other systems, you must redefine these words to work in your particular environment. Also, the words on screen 7 are specific to the IBM PC; they control the serial port, so you will also need to revise them for other machines.

The glossary that follows is a word-by-word description of the routines presented in Listing One (page 74). The words appear in the same order as they do in the listing. The glossary is intended to help you understand what's been implemented, so you can modify the routines to best suit your needs.

Glossary For SEND/RCV Listing

Screen 1: These are various words that extend PC/FORTH.

0<> n – f
If n<>0, then f=1, else f=0.

<> n1 n2 – f
If n1=n2, then f=1, else f=0.

>= n1 n2 – f
If n1>=n2, then f=1, else f=0.

<= n1 n2 – f
If n1<=n2, then f=1, else f=0.

NOT n1 – n2
n2 is the logical complement of n1.

RESTART
Compiler word. Usage: RESTART
Used inside BEGIN ... AGAIN or BEGIN ... UNTIL loops to start them over from the beginning.

Screens 2-6: This is a PCDOS file handler, defined in terms of a PC/FORTH primitive word FDOS that provides low-level access to the PCDOS operating system calls. Although these screens are operating system dependent, they are not machine dependent. Thus, they will work with PCDOS on machines other than an IBM PC.

FCB
Defining word. Usage: FCB name
Constructs a PCDOS file control block. After "name" is defined to be an FCB, it will push the FCB's address on the stack.

SET-DMA fcb-addr –
Sets the buffer area for file transfers involving the specified FCB.

?BUFFER-ADDR fcb-addr – fcb-offset
Pushes the location of the specified FCB's buffer area.

OPEN-FILE fcb-addr – status
Status=0 if the file indicated by the FCB was successfully opened; any error, and status=255.

CLOSE-FILE fcb-addr – status
Same as OPEN-FILE, only closing the file.

MAKE-FILE fcb-addr – status
Same as OPEN-FILE, only creating a new file.

DELETE-FILE
Not used by SEND or RCV.

SEARCH-FILE
Not used by SEND or RCV.

NEXT-FILE
Not used by SEND or RCV.

RENAME-FILE
Not used by SEND or RCV.

READ-SEQ fcb-addr – status
Reads the next block from the file indicated by the FCB; status=0 if the operation is a success. (Note that this is different than OPEN-FILE, where a status of 0 indicated an error.)

WRITE-SEQ fcb-addr – status
Writes the block from the buffer to the file indicated by the FCB; status=0 if the operation is a success.

READ-RANDOM
Not used by SEND or RCV.

WRITE-RANDOM
Not used by SEND or RCV.

PARSE fcb-addr buf-ptr – f1
Looks at the file specification stored in the buffer pointed to by buf-ptr and moves it into the FCB, properly formatted. The return code is f1=0 if the file spec is OK, f1=1 if the file spec contains wildcards, and f1=255 if the file spec is not valid.

FILENAME fcb-addr – f1
Usage: fcb FILENAME name
Parses a file specification from the input stream into the FCB at fcb-addr. f1 is returned the same way as in PARSE: f1=0 is an OK spec, f1=1 is a wildcard spec, f1=255 is an invalid spec.

IN-NAME fcb-addr – f1
Prompts for a file specification, which gets put into the FCB at fcb-addr. f1 is returned as in PARSE.

OUT-NAME fcb-addr –
Displays the file name that is contained in the FCB at fcb-addr.

Screen 7: This is the only machine dependent screen in the entire listing. These words provide low-level serial port support: send a byte, receive a byte, check to see if a byte has arrived, etc.

?COM – f
If f=1, a byte is waiting at the serial port to be read.

WAITOUT –
Loops until the serial port can accept a byte to be transmitted.

WAITIN –
Loops until a byte is waiting at the serial port to be read.

XIN – n
Reads a byte from the serial port.

XOUT n –
Writes a byte to the serial port. If n=-1, nothing is written to the port.

CLEAR
Waits until there are no characters waiting

to be read.

I300 –

Initializes the serial port to 300 baud, 8 bits are transmitted, no parity, 1 stop bit, and turns on the DTR signal. If you want to change the baud rate, the HEX 180 should be changed as follows:

To Get:	*Use HEX:*
300 baud	180
1200 baud	60
4800 baud	18
9600 baud	C

Screen 8: These are the variables and constants used by SEND and RCV.

LONG-TIMEOUT –

Sets the timer's initial value to 30000, so the timer will wait a relatively long time before generating a timeout error.

SHORT-TIMEOUT –

Sets the timer's initial value to 10000, so the timer will wait a relatively short time before generating a timeout error.

Screens 9-13: These are the RCV and auxiliary routines.

RCV-SETUP –

Creates the file to be received. If the file already exists, asks if it's OK to overwrite the existing file with the incoming file, then sets a few variables.

TIMED-IN – n

Waits a predetermined time for a byte to be received. If nothing shows up during that time, then n=-1, and we say a time-out has occurred.

GOBBLE –

Ignores all incoming characters until no characters arrive for a certain period of time; that is, GOBBLE waits until a time-out is generated.

ABEND –

In case of an abnormal end, ABEND informs you and closes the file that is being sent/received.

RCVERR –

Invokes GOBBLE to clear the line, then sends a NAK to indicate that the receiver has detected an error. If more than MAXERR errors have occurred, the transmission is aborted.

?TERMABORT – f

f=-1 if Ctrl-X has been typed on the keyboard (typing Ctrl-X is a manual cancel of the transmission).

NOREND –

Normal end that closes the file being sent or received.

RCV-HDR – f

Gets the standard XMODEM sector header and checks it for validity; f=-1 if the header is incorrect.

RCV-SECT – f

Gets the sector and computes a checksum; f=-1 if there is an error.

RCV-CKSUM – f

Gets the checksum sent by the sender and compares it with the computed checksum; f=-1 if they don't match.

SAVE-SECT –

Writes the sector from the buffer into the file on disk.

DISP-CHR n –

Displays unexpected characters.

RCV –

Usage: RCV filespec.ext

Receives the file specified from the serial port using the XMODEM protocol (see the text of the article).

Screens 14-17: These are the SEND and auxiliary routines.

WAIT-ACK –

Waits for an ACK, a NAK, a CAN, a time-out, or a manual cancel (CTRL-X).

READ-SECT – f

Reads the next sector to be transferred from the disk file into the buffer; f=-1 if End Of File.

SEND-HDR –

Transmits a sector header.

SEND-SECT –

Transmits a sector and computes a checksum.

SEND-CKSUM –

Transmits the checksum computed by SEND-SECT.

END-SEND –

Transmits an EOT and waits until it gets an ACK from the receiver, then closes the file being sent.

SEND-SETUP –

Initializes the serial port, inputs the file specification, opens the file, and reports on the file's size.

SEND –

Usage: SEND filespec.ext

Transmits the file specified to the serial port using the XMODEM protocol (see the text of the article).

DDJ

(Listing begins on page 74)

SEND and RCV - Listing

(Text begins on page 66)

```
Screen # 0
( SEND and RCV files using XMODEM protocol          06/03/83 )

Copyright 1982, 1983 by Robert Taylor III
All rights reserved.
These routines are not to be used for commercial purposes,
but unlimited personal, non-commercial use is allowed.

For a full communication package that includes these routines,
please order "PC/TERM" from:
    Laboratory Microsystems
    4147 Beethoven St.
    Los Angeles, CA 90066

PC/TERM is an extension package, designed to be run with
Laboratory Microsystems' PC/FORTH.  It is best used with a
Hayes Smartmodem.
```

```
Screen # 1
( SEND and RCV files using XMODEM protocol          06/03/83 )

( various supplementary arithmetic functions                  )
: 0<> 0= 0= ;   : <> = 0= ;     : >= < 0= ;     : <= > 0= ;
: NOT -1 XOR ;

( extension to FORTH compiler: used inside a BEGIN loop to    )
( start the loop over from the beginning. use:                )
(   BEGIN    xx IF RESTART ENDIF     AGAIN                    )
: RESTART ( ; use in BEGIN loops to start them over)
  ?COMP ?STACK  0 0 >R >R
  BEGIN  DUP 1 <> WHILE >R >R REPEAT
  2DUP [COMPILE] AGAIN
  BEGIN R> R> 2DUP 0 0 D= UNTIL 2DROP ; IMMEDIATE
-->
```

```
Screen # 2
( SEND and RCV files using XMODEM protocol          06/03/83 )

FORTH DEFINITIONS DECIMAL
( used in the form:  FCB fcb-name )
: FCB           <BUILDS  HERE 165 ERASE 165 ALLOT
                DOES> ;

( length of fcb, displacement to actual i/o buffer )
37 CONSTANT BUFFER-OFFSET

( fcb-addr  ---  )
: SET-DMA       26 SWAP BUFFER-OFFSET + FDOS 2DROP ;
( fcb-addr  ---  )
: ?BUFFER-ADDR  BUFFER-OFFSET + ;
-->
```

```
Screen # 3
( SEND and RCV files using XMODEM protocol          06/03/83 )

( for all file operations, status code = 0FFh if )
( operation failed, zero if operation successful. )

( fcb-addr  ---  status-code )

: OPEN-FILE     15 SWAP FDOS DROP 255 AND ;
: CLOSE-FILE    16 SWAP FDOS DROP 255 AND ;
: MAKE-FILE     22 SWAP FDOS DROP 255 AND ;
: DELETE-FILE   19 SWAP FDOS DROP 255 AND ;
: SEARCH-FILE   17 SWAP FDOS DROP 255 AND ;
: NEXT-FILE     18 SWAP FDOS DROP 255 AND ;
: RENAME-FILE   23 SWAP FDOS DROP 255 AND ;
-->
```

```
Screen # 4
( SEND and RCV files using XMODEM protocol          06/03/83 )
( for all record operations, status code = buffer address )
( if operation successful, zero if operation failed. )

( fcb-addr  ---  status-code For READ-SEQ, partial = success )
: READ-SEQ      DUP DUP SET-DMA 20 SWAP FDOS DROP 255 AND 1 =
                IF DROP 0 ELSE BUFFER-OFFSET + THEN ;
: WRITE-SEQ     DUP DUP SET-DMA 21 SWAP FDOS DROP 255 AND 0=
                IF BUFFER-OFFSET + ELSE DROP 0 THEN ;

( fcb-addr  record-number  ---  status-code )
: READ-RANDOM   OVER 33 + ! DUP DUP SET-DMA 33 SWAP FDOS DROP
      255 AND 0= IF BUFFER-OFFSET + ELSE DROP 0 THEN ;
: WRITE-RANDOM  OVER 33 + ! DUP DUP SET-DMA 34 SWAP FDOS DROP
      255 AND 0= IF BUFFER-OFFSET + ELSE DROP 0 THEN ;
-->
```

```
Screen # 5
( SEND and RCV files using XMODEM protocol          06/03/83 )
BASE @ HEX
( use PARSE to analyze a typed-in file specification.         )
CODE PARSE ( fcb-addr buf-ptr --- fl; 0=OK, 1=WILD, 255=BAD    )
    AX POP   DX POP   BP PUSH   SI PUSH   ES PUSH
    DI, DX MOV   DX, DS MOV   ES, DX MOV   SI, AX MOV
    AX, # 2901 MOV     # 21 INT    AX, # 00FF AND
    ES POP   SI POP   BP POP   AX PUSH   NEXT
BASE !


-->
```

```
Screen # 6
( SEND and RCV files using XMODEM protocol          06/03/83 )

: FILENAME      ( fcb-addr  --- fl; fl:0=OK,1=wild,255 not OK )
                BL WORD HERE 1+ PARSE   ;

: IN-NAME       ( fcb-addr  --- fl; fl:0=OK,1=wild,255 not OK )
                ." Enter file specification: "
                HERE 30 EXPECT HERE PARSE   ;

: OUT-NAME      ( fcb-addr --- )
  DUP C@ -DUP IF 64 + EMIT 58 EMIT  ( : ) ENDIF
  9 1 DO DUP I + C@ DUP 32 = IF DROP LEAVE ELSE EMIT ENDIF LOOP
  46 EMIT ( . )
 12 9 DO DUP I + C@ DUP 32 = IF DROP LEAVE ELSE EMIT ENDIF LOOP
  DROP ;
-->

Screen # 7
( SEND and RCV files using XMODEM protocol          06/03/83 )
HEX
: ?COM 3FD PC@ 1 AND ;                  ( char available?     )
: WAITOUT BEGIN 3FD PC@ 20 AND UNTIL ;  ( wait for TX empty   )
: WAITIN BEGIN ?COM UNTIL ;             ( wait for char avail )
: XIN  3F8 PC@ ;                        ( receive a char      )
: XOUT 1+ -DUP IF WAITOUT 1- 3F8 PC! THEN ; ( TX 1 chr, exc.-1)
: CLEAR ?COM IF XIN DROP ENDIF ;        ( clear line garbage  )
: I300 80 3FB PC! 180 3F8 P!  3 3FB PC! 1 3FC PC! ;
DECIMAL
-->

Screen # 8
( SEND and RCV files using XMODEM protocol          06/03/83 )
  FCB XFER              ( FCB area where file is opened       )
0 VARIABLE CKSUM        ( check sum byte for a 128 byte sector )
( error variables )
0 VARIABLE ERRS   10 CONSTANT MAXERR    0 VARIABLE NAK-CT
( various sector variables: rcv sec, send sec, expected sec   )
0 VARIABLE RSEC    0 VARIABLE SSEC      0 VARIABLE XSEC
( ascii characters with special meaning in the XMODEM protocol)
0 CONSTANT NUL    24 CONSTANT CAN       1 CONSTANT SOH
4 CONSTANT EOT     6 CONSTANT ACK      21 CONSTANT NAK
( timer control: TIM-CT is the timer's initial setting.       )
10000 CONSTANT TIM-CT   0 VARIABLE TIMER
: LONG-TIMEOUT  30000 ' TIM-CT ! ;      ( longer timeout dur. )
: SHORT-TIMEOUT 10000 ' TIM-CT ! ;      ( shorter timeout dur.)
-->

Screen # 9
( SEND and RCV files using XMODEM protocol          06/03/83 )

: RCV-SETUP
  I300
  XFER FILENAME  DROP CR  XFER OPEN-FILE  0=
  IF    ." File exists. OK to replace? " PCKEY CR
        -DUP 0 <> # 223 AND 89 <>
        IF QUIT ENDIF
        XFER CLOSE-FILE DROP
  ENDIF
  XFER MAKE-FILE
  IF    ." Can't create file: "  XFER OUT-NAME CR QUIT
  ENDIF
  XFER OUT-NAME  ."  created and opened." CR
  0 XFER 32 + C!  0 RSEC !   0 XSEC !   0 NAK-CT !     ;
-->

Screen # 10
( SEND and RCV files using XMODEM protocol          06/03/83 )
: TIMED-IN  TIM-CT TIMER !
  BEGIN  ?COM IF XIN EXIT ENDIF   -1 TIMER +!   TIMER @ 0=
  UNTIL -1 ;

: GOBBLE BEGIN TIMED-IN -1 = UNTIL ;

: ABEND   ." Aborting." CR  XFER CLOSE-FILE
  IF ." Can't close file: " XFER OUT-NAME CR ENDIF ;

: RCVERR  GOBBLE 1 NAK-CT +!  NAK-CT @ MAXERR >
          IF   CAN XOUT  ."  Exceeded maximum errors." CR
               ABEND  QUIT
          ENDIF
          ." NAK sent." CR  NAK XOUT  ;
-->

Screen # 11
( SEND and RCV files using XMODEM protocol          06/03/83 )

: ?TERMABORT 0 ?TERMINAL IF DROP KEY 24 =
  IF ." Cancelled at console." CR -1 ENDIF ENDIF ;

: NOREND ." EOT Received." CR  ACK XOUT  XFER CLOSE-FILE
  IF ." Can't close file: " XFER OUT-NAME CR ENDIF  ;

: RCV-HDR  TIMED-IN DUP -1 = IF EXIT ENDIF DUP RSEC !
  TIMED-IN DUP -1 = IF EXIT ENDIF
  255 XOR <> IF -1 ELSE 0 ENDIF ;

: RCV-SECT 0 CKSUM C!  128 0 DO TIMED-IN DUP -1 = IF EXIT ENDIF
       DUP XFER ?BUFFER-ADDR I + C!  CKSUM C@ + CKSUM C!
  LOOP 0 ;
-->

Screen # 12
( SEND and RCV files using XMODEM protocol          06/03/83 )

: RCV-CKSUM  TIMED-IN DUP -1 =
             IF EXIT ENDIF
             CKSUM C@ =
             IF 0 ELSE -1 ENDIF ;

: SAVE-SECT  XFER WRITE-SEQ  0=
             IF ." Can't write sector." CR ABEND QUIT
```

(Continued on page 81)

SEND and RCV - Listing

(Listing continued, text begins on page 66)

```
            ENDIF
            XSEC @ . ." written." CR  1 XSEC +! ;

: DISP-CHR   BASE @ >R CR DUP HEX 2 .R R> BASE !
            ." H(" EMIT ." ) unexpectedly seen. " ;
-->
```

Screen # 13

```
( SEND and RCV files using XMODEM protocol          06/03/83 )

: RCV   RCV-SETUP
  GOBBLE ." Initial NAK sent." CR    NAK XOUT
  BEGIN  ?TERMABORT IF ABEND QUIT ENDIF  ( use Ctrl-X to abort )
    TIMED-IN
      DUP NUL =  IF DROP RESTART ENDIF
      DUP CAN =  IF DROP ." Remote Cancel." CR ABEND QUIT ENDIF
      DUP EOT =  IF DROP NOREND QUIT ENDIF
      DUP -1  =  IF DROP ." T-" RCVERR RESTART ENDIF
      DUP SOH <> IF DISP-CHR RCVERR RESTART ENDIF    DROP
    RCV-HDR   IF ." H-" RCVERR RESTART ENDIF
    RCV-SECT  IF ." S-" RCVERR RESTART ENDIF
    RCV-CKSUM IF ." C-" RCVERR RESTART ENDIF
    SAVE-SECT  ACK XOUT   0 NAK-CT !
  AGAIN ;   -->
```

Screen # 14

```
( SEND and RCV files using XMODEM protocol          06/03/83 )
: WAIT-ACK   0 ERRS !   ( -- ; wait for ACK or CAN, w/ timing )
            BEGIN    ERRS @ MAXERR > NAK-CT @ MAXERR > OR
              IF ." Exceeded Maximum errors." CR CAN XOUT
                 ABEND QUIT ENDIF
              ?TERMABORT IF ABEND QUIT ENDIF   ( ^X ? )
              TIMED-IN  CASE
                 -1 OF 1 ERRS +! ." Timeout" CR ENDOF
                CAN OF ." Remote Cancel" CR ABEND QUIT ENDOF
                ACK OF 0 NAK-CT !  EXIT ENDOF
                NAK OF 1 NAK-CT +! EXIT ENDOF
              ( ELSE)  DISP-CHR 0 ENDCASE
            AGAIN ;

: READ-SECT  SSEC @ XFER 32 + !  XFER READ-SEQ 0= ;
-->
```

Screen # 15

```
( SEND and RCV files using XMODEM protocol          06/03/83 )

: SEND-HDR  ( -- ; send header: SOH sec-# NOT[sec-#] )
            SOH XOUT
            SSEC @ 1+ 255 AND DUP XOUT  255 XOR XOUT
            ." Sending " SSEC @ . CR ;

: SEND-SECT ( -- ; send each byte of 128 byte sector. )
            0 CKSUM C!
            128 0 DO XFER ?BUFFER-ADDR I + C@ DUP XOUT
                CKSUM C@ + CKSUM C!  LOOP ;

: SEND-CKSUM  CKSUM C@ XOUT ;  ( -- ; cksum=SUM[all 128 bytes])
-->
```

Screen # 16

```
( SEND and RCV files using XMODEM protocol          06/03/83 )
: END-SEND      ( -- ; terminate transmission, wait for ACK   )
  XFER CLOSE-FILE DROP
  BEGIN  CR ." Sending EOT -" EOT XOUT  WAIT-ACK
         NAK-CT @ 0=
  UNTIL  ." ack'd" CR ;

: SEND-SETUP
  I300    LONG-TIMEOUT
  XFER FILENAME  DROP CR  XFER OPEN-FILE
  IF   ." File doesn't exist. Exitting." CR QUIT
  ENDIF
  XFER OUT-NAME ."  opened. Size is: " XFER 16 + 2@ SWAP 2DUP D.
                ."  (" 128 M/ SWAP 0 <> + . ." sectors)" CR
  128 XFER 14 + !  0 SSEC !  0 NAK-CT !   ;
-->
```

Screen # 17

```
( SEND and RCV files using XMODEM protocol          06/03/83 )

: SEND          ( -- ; uses XMODEM protocol to send a file    )
  SEND-SETUP                                    ( get file rdy)
  SHORT-TIMEOUT   GOBBLE  LONG-TIMEOUT          ( clear line  )
  ." Awaiting Initial NAK: "
  WAIT-ACK  CR  SHORT-TIMEOUT                   ( wait for NAK)
  BEGIN ?TERMABORT IF ABEND QUIT ENDIF          ( Ctrl-X hit? )
        READ-SECT  IF END-SEND QUIT ENDIF   ( at end of file? )
        SEND-HDR                              ( send header )
        SEND-SECT                             (  "   sector )
        SEND-CKSUM                            (  " checksum )
        WAIT-ACK      NAK-CT @ 0=             ( ACK rcv'd?  )
        IF 1 SSEC +! ENDIF                  ( yes, incr. Sec #)
  AGAIN ;
```

End Listing

Precompiled Forth Modules

by N. Solntseff

N. Solntseff, Unit for Computer Science, McMaster University, Hamilton, Ontario L8S 4K1.

A Forth compilation is a relatively slow process. In the system I use – a standard Fig-Forth78* running on an old OSI video machine – my full-screen editor occupies 35 screens and takes more than three minutes to load and compile. This is intolerably long for production programs. As a result, I have developed another way of loading programs through the use of precompiled modules. The main idea behind my approach is to treat a precompiled module as a separate vocabulary. Thus, each overlay has a header (analogous to a vocabulary header) that contains the information needed by the module loader for linking the module dictionary with the Forth dictionary.

The source code for the words implementing overlays is contained in Listing One (page 84). I should mention here that my editor includes a command for calculating checksums as described by Suralis and Brodie.[1] The CRC-16 check value is automatically inserted in line 0 of each screen. I have also followed my own recommendations concerning Forth programming style.[2]

The header consists of a vocabulary pseudo name-field 81A0 followed by the link field, which contains the name-field address of the topmost Forth word at the time the overlay header is created (the value of LATEST). Next come the number of screens occupied by the compiled overlay words, the value of FENCE for the overlay, the name-field address of the topmost overlay word, and finally the address of the top of the dictionary including the overlay. Overlay headers are created by OVSTART (Screen 31). The accompanying figure (page 83) illustrates the function of the overlay header and shows the dictionary configuration after an overlay has been loaded into memory.

To prepare an overlay, the user calls MAKE-OVERLAY (Screen 35), supplying the number of the first screen containing the source code of the program to be made into an overlay. This creates an overlay header, after which an <n> LOAD is executed to load and compile the program into memory. Either Screen <n> should be a master screen, which loads the rest of the program, or the screens should be linked together by means of the word -->. After the program is loaded, OVEND (Screen 32) determines how many screens of binary information are needed to store the overlay and completes the header by storing the values of FENCE, HERE, and LATEST in the appropriate header locations. After preparing an overlay, the user can store it on disk by invoking <n> PUT (Screen 33). The responsibility of ensuring that sufficient empty screens are available on disk for storage is left to the user. PUT checks the screens to see if, in fact, they contain overlay code and aborts if this is not the case. The check is performed by the word ?OV.

Once an overlay module is stored on disk, it can be loaded ready for execution by means of the word GET (Screen 34). GET loads several consecutive screens after checking that the first screen is indeed an overlay preceded by an overlay header; GET aborts if this is not the case. Its last action is to set the vocabulary pointer CURRENT and FENCE to protect the loaded overlay. GETS is a convenience word that can be used to load specific overlays by name. For example, I have defined the word EDITOR to load my full-screen editor so that I do not have to remember the numbers of the screens containing its overlay. An overlay is easily removed by invoking the routine EMPTY (Screen 30), which clears the user dictionary down to the original Forth system portion.

The overlay facility has proved to be indispensable. The editor now loads in under three seconds – a reduction in loading time approximately by a factor of 60! A word of caution is needed here: Once you have loaded and compiled Screens 30-35, you must make all the overlay words a permanent part of the Forth system, so that they are available after a cold start and, more importantly, are protected by the system FENCE. This precaution is necessary because overlays always start immediately after the Forth system part of the dictionary. As this procedure is highly operating system dependent, I will leave you to figure out what must be done for your operating system. Finally, I would recommend that the word .LINE be redefined if overlays are used extensively, to prevent binary bytes from being treated as control characters by the display and printer drivers.

Glossary

?OV <addr> –

Issue an error message (number 9) if the location with address <addr> does not contain the start of an overlay header. Execution of the word containing ?OV is aborted.

EMPTY –

Clear the user part of the dictionary down to the topmost Forth definition. Both the value of FENCE and the pointer to the top name in the word FORTH are reset to their cold-start values.

FWAUD – <addr>

Leave the address of the start of the user dictionary on the data stack. This is the starting address of all overlays.

GET <n> –

Read a sequence of binary screens starting with Screen <n> and place the overlay into memory starting at the location given by FWAUD.

GETS

A defining word used in the form

<n> GETS <name>

**I have stuck to the original Fig model and implementation because, in my experience, extensions and improvements made by implementors introduce non-trivial problems due to the resulting lack of portability. Of the half-dozen versions of Forth for OSI and Commodore 64 computers I have acquired, all claiming full compatibility with the Fig-Forth79 standard,* none *are 100% consistent with the latter. In my opinion, it is better to have a completely specified model, such as the Fig-Forth78 one, and an implementation that rigorously adheres to it, than an improved model with implementations that vary among themselves. This avoids the need for detective work to establish the effect of allegedly standard words. In the case of the Fig-Forth78 model, a distributor of an application vocabulary need only supply the additional words needed to extend his or her system to the required level.*

to create a word <name>. When <name> is later executed, it performs an

<n> GET

to load an overlay from a sequence of screens starting with screen <n>.

MAKE-OVERLAY <n> –

LOAD a sequence of screens starting with screen <n>, provided the screens are linked by the word --->. First, the user dictionary is emptied, and an overlay header is inserted at the front. After the source screens are compiled, the overlay header is completed with appropriate values. The overlay can be stored on disk in one or more screens by means of the word PUT.

NTOP – <addr>

Leave the name-field address of the last word in the Forth system dictionary. This is used for linking the overlay with Forth.

OVEND –

An auxiliary word used to complete the overlay header after a Forth program has been LOADed as an overlay.

OVSTART –

An auxiliary word used to initialize the overlay header. Used by MAKE-OVERLAY.

PUT <n> –

Write a sequence of binary screens starting with screen <n> to store an overlay whose starting address is given by FWAUD.

Notes

[1] K. Suralis and L. Brodie, Sept./Oct. 1982, "Checksum for Hand-Entered Source Screens," *Forth Dimensions,* 4:15.

[2] N. Solntseff, Sept. 1982, "Forth Programming Style," *Dr. Dobb's Journal,* 7:30-32.

DDJ

(Listing begins on page 84)

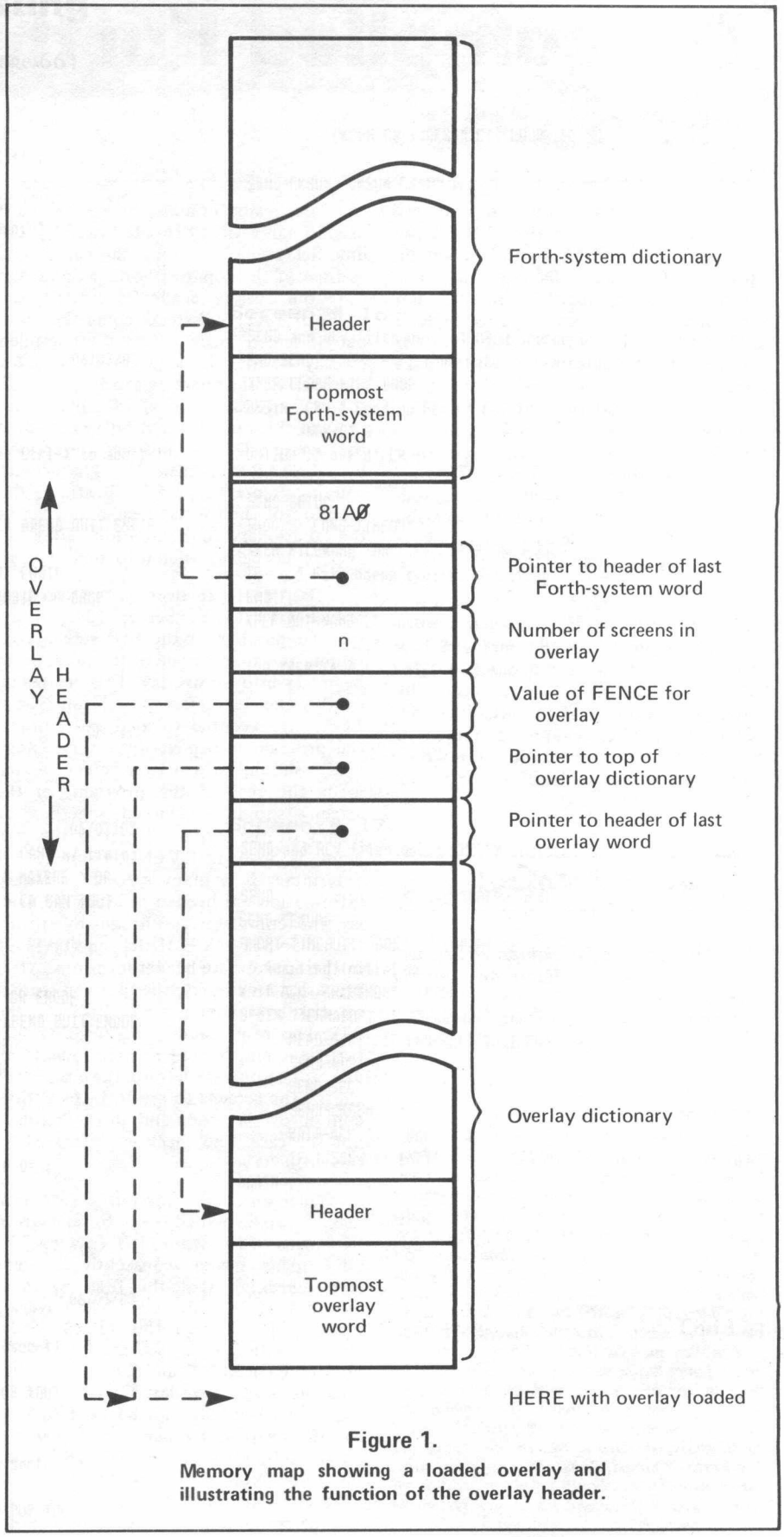


Figure 1.

Memory map showing a loaded overlay and illustrating the function of the overlay header.

Precompiled Forth Modules

(Text begins on page 82)

```
SCR # 30
  0 ( 40801 -- overlays -- FWAUD  NTOP  ?OV  EMPTY   30/ NS/10may83)
  1
  2 FORTH DEFINITIONS HEX
  3
  4 : FWAUD 1C +ORIGIN @ ;     ( address of start of user dictionary)
  5 : NTOP  0C +ORIGIN @ ;       ( address of last word in empty UD)
  6
  7 : ?OV            ( addr -- ; check if an overlay begins at addr)
  8 ( addr) @ A081 - 09 ?ERROR ;
  9
 10 : EMPTY   ( -- ; clear user dictionary up to topmost FORTH word)
 11           [COMPILE] FORTH DEFINITIONS
 12           NTOP ' FORTH 4 + !       ( reset NTOP in FORTH vocab.)
 13           FWAUD DUP FENCE !     ( reset FENCE to original value)
 14  ( fwaud) DP ! ;                    ( reset dictionary pointer)
 15  -->
```

```
SCR # 31
  0 ( 60325 -- overlays -- OVSTART                    31/ NS/10may83)
  1
  2 : OVSTART                        ( -- ; initialize overlay header)
  3         EMPTY HERE                ( remove all user-defined words)
  4  ( top) A081    ,              ( create a dummy vocabulary header)
  5  ( top) LATEST ,                ( save nfa of topmost FORTH word)
  6  ( top) CURRENT @ !                    ( reset vocabulary pointer)
  7         0000    ,         ( space for number of screens in overlay)
  8         0000    ,           ( space for value of FENCE for overlay)
  9         0000    ,       ( space for top of dictionary for overlay)
 10         0000    , ;       ( space for nfa of topmost overlay word)
 11
 12 -->
 13
 14
 15
```

```
SCR # 32
  0 ( 30432 -- overlays -- OVEND                      32/ NS/10may83)
  1
  2 : OVEND       ( -- ; complete overlay header after program LOAD)
  3                CR FWAUD          ( get starting addr of overlay)
  4          ( fwa) HERE FWAUD - B/BUF /MOD ( find number of screens)
  5   ( fwa n rem) SWAP IF 1+ ENDIF                  ( adjust number)
  6    ( fwa #scr) DUP . ." screens required by overlay" CR
  7    ( fwa #scr) OVER 4 + !                              ( and store)
  8         ( fwa) FENCE @ OVER  6 + !        ( store overlay  FENCE)
  9         ( fwa) HERE    OVER  8 + !        ( store overlay   HERE)
 10         ( fwa) LATEST  OVER 0A + !        ( store overlay LATEST)
 11         ( fwa) DROP ;
 12
 13 -->
 14
 15
```

```
SCR # 33
  0 ( 40998 -- overlays -- OVSAVE                          33/ NS/11may83)
  1
  2 : PUT       ( n -- ; save overlay to disc starting with screen n)
  3             ( n) FWAUD ?OV              ( get fwa of save & check)
  4             ( n) FWAUD 4 + @          ( get no of screens needed)
  5         ( n #bl) 0 DO                          ( repeat over blocks)
  6             ( n)       DUP FWAUD I B/BUF * +    ( get from address)
  7        ( n from)       OVER I + DUP . OFFSET @ +    ( form block #)
  8   ( n from scr#)       BUFFER                     ( get to address)
  9     ( n from to)       B/BUF CMOVE           ( move to disc buffer)
 10             ( n)       UPDATE FLUSH                ( write to disc)
 11             ( n)    LOOP              ( until all screens are read)
 12             ( n) DROP CR ;
 13 -->
 14
 15
```

```
SCR # 34
  0 ( 41428 -- overlays -- OVLOAD                          34/ NS/11may83)
  1
  2 : GET               ( n -- ; load overlay starting with screen n)
  3         ( n) DUP .                              ( echo screen number)
  4         ( n) DUP BLOCK DUP ?OV        ( read screen n into buffer)
  5   ( n bufad) DUP 8 + @ DP !               ( make room for overlay)
  6   ( n bufad) DUP FWAUD B/BUF CMOVE              ( move first block)
  7   ( n bufad) 4 + @ 1 - -DUP     ( get number of scfeens and check)
  8   ( n #sc-1) IF                          ( more than one screen used)
  9         ( n)     1+ 1 DO DUP I + DUP . BLOCK          ( read screen)
 10    ( n from)              I B/BUF * FWAUD + B/BUF CMOVE      ( move)
 11         ( n)           LOOP DROP           ( until all screens read)
 12              ENDIF CR
 13              FWAUD 0A + @ CURRENT !      ( set vocabulary pointer)
 14              FWAUD  6 + @ FENCE   ! ;         ( set overlay FENCE)
 15 DECIMAL -->
```

```
SCR # 35
  0 ( 14136 -- overlays -- MAKE-OVERLAY PUT GET GETS 35/ NS/11may83)
  1
  2  : MAKE-OVERLAY       ( n -- ; LOAD screen n and prepare overlay)
  3    ( n) OVSTART LOAD OVEND ;
  4
  5  : GETS              ( defining word: n GETS <name>              )
  6                      ( at run time, <name> performs an   n GET  )
  7       <BUILDS  ,
  8         DOES>  @ GET ;
  9
 10
 11
 12
 13
 14
 15
```

End Listing

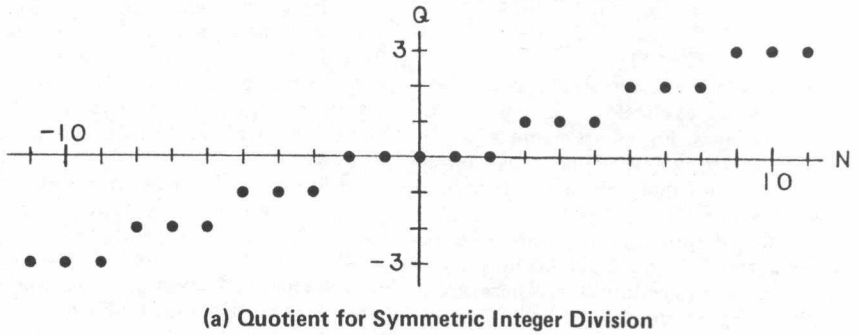
Signed Integer Division

by Robert L. Smith

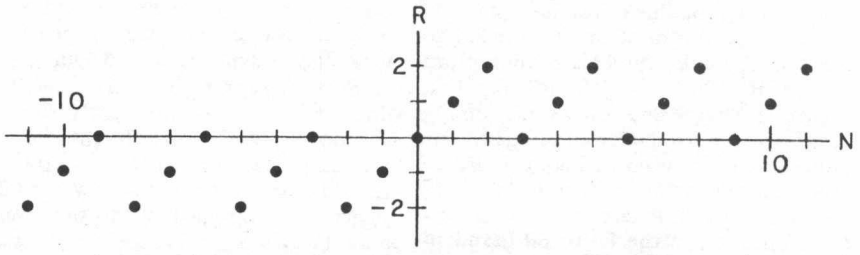
Not all methods of integer division produce a uniform result when the dividend and divisor have opposite signs. This may not be so important in the area of commerce where negative values are perhaps used less. When dealing with measurement and control, however, uniformity becomes more significant. The Forth 83 Standard adopts a method for *signed integer* division called "floored" division. While APL has used this method for the RES function for years and Small-Talk provides it as one of three methods for integer division, acceptance of the Standard makes Forth the first "popular" language to embrace this method based on its theoretical merits. The problem is one of mathematical purity versus user expectation. This article will attempt to clarify some of the issues involved.

Integer division is a mathematical function of two integers (a dividend and a divisor) that yields an integer quotient and an integer remainder. That appears to be a fairly straightforward operation, but there is not universal agreement of the desired results when one or both arguments are negative. When an integer quotient is used in plotting or machine control, the desired function is usually *not* the quotient given by the majority of computers.

Most computers with a divide function produce a quotient that has a property of symmetry around zero when plotted as a function of the dividend, due to the fact that the quotient is rounded toward zero. Speaking mathematically, the property is actually one of anti-symmetry, where the sign of the quotient is reversed when the sign of the dividend (or numerator) is reversed. For integer division, this "symmetric" property leads to a sort of discontinuity around zero. In this case, the remainder is either zero or it takes the sign of the dividend. Figure 1a (at right) illustrates the quotient q as a function of a variable dividend, and a constant divisor 3. We readily see the discontinuity near zero. This may be reasonably serious when this quotient function is used for plotting or moving robot arms.

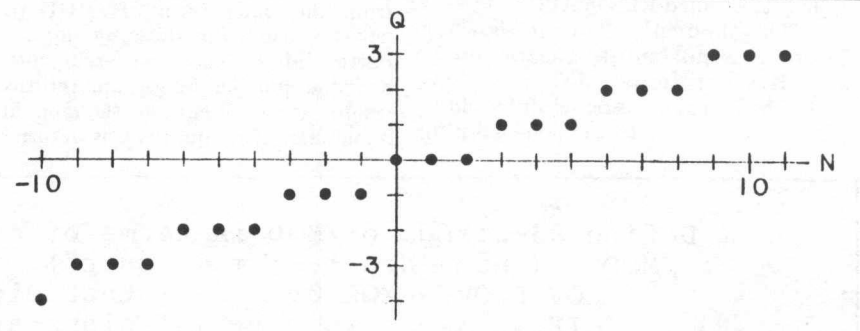


(a) Quotient for Symmetric Integer Division

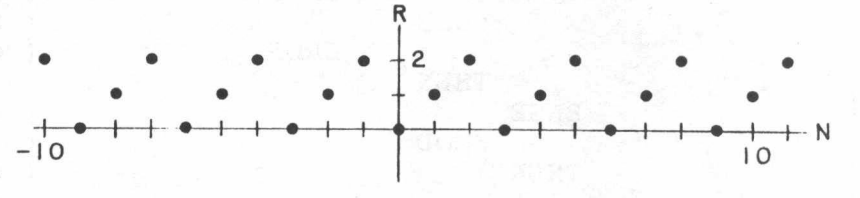


(b) Remainder for Symmetric Integer Division

Figure 1.



(a) Quotient for Floored Integer Division



(b) Modulus Function for Floored Integer Division

Figure 2.

Robert L. Smith, ESL, Inc., 495 Java Dr., Sunnyvale, California 94086.

The integer quotient needs an associated remainder:

```
r = n - q*d
```

where n is the numerator or dividend, d is the denominator or divisor, q is the quotient, and r is the remainder. The remainder function for the constant divisor 3 is illustrated in Figure 1b (page 86). If we look at the case of positive dividends and *divisors*, we observe the cyclic property that

```
r(n+d) = r(n)
```

In other words, the remainder usually has a repeating or cyclical property as the dividend changes. For the remainder shown in Figure 1b, we see that this simple property is not maintained for dividends between -d and 0.

If we require that the remainder be cyclical, then the quotient no longer has any unusual discontinuities. There are a number of possible choices here. One obvious choice is to make the remainder the same as the modulus or residue function.[1] In this case the quotient is rounded toward minus infinity. This rounding procedure is called the "floor" function. Figure 2 (page 86) shows the floored quotient and its related modulus for the same arguments used in Figure 1. Notice the quotient behaves in a more nearly continuous fashion around zero. This is the form used in the Forth-83 Standard, as well as some of the older versions of Forth. The National 16032 microprocessor produces floored division in addition to the older "rounded toward zero" variety. The modulus function is called MOD in Forth-83 and in the National 16032. It is called RES in APL.

The "floored" quotient shown in Figure 2 is not anti-symmetric around zero. However, for odd divisors one may easily obtain a symmetric result by adding a correction factor to the dividend prior to division. Although the quotient is generally not defined when the divisor is zero, the modulus is usually defined to take the value of the dividend for this case. If infinities are not allowed in computer representations, and the product of any number and zero is always zero, then this definition preserves the equation

```
n = q*d + r
```

for all values of d, including zero.

Alternative remainder functions include a positive modulus and a remainder that takes the sign of the quotient.[2] Some other possibilities have the undesirable feature of negative remainders when the dividend and divisor are both positive.

Floored division is simply more useful in the majority of applications programs. The major objection is that the results are not what most people expect: -1 divided by 4 gives 0 in the rounded-toward-zero division case, but -1 for floored division. Both cases give the same results when the dividend and divisor have the same sign. Timing efficiencies may play a small role in deciding which form of division to use, but generally the division process is sufficiently slow that additional tests for different forms of rounding take only a little extra time. Indeed, for some processors with built-in signed and unsigned divide functions, it may be faster in the common case of positive arguments to test signs and use the unsigned division than to just use the signed division function. If you have an older Forth system (such as 79-Standard or Fig-Forth), the screen in Figure 3 (page 88) shows a high-level conversion from the older form of /MOD to the newer version. For those unfamiliar with Forth, /MOD takes two arguments, the dividend and the divisor, and returns two results: the quotient and the modulus, or remainder. The quotient is returned as the most accessible element on the stack.

The appearance of floored division in some of the newer processor chips and languages indicates the increasing awareness of its utility. We might note in passing that even floating-point division will probably be different in the future than it was in the past due to the new Floating-Point Standard, which will require proper rounding of the quotient.

References

[1] Donald K. Knuth, *The Art of Computer Programming: Volume 1, Fundamental Algorithms,* Second Edition, Menlo Park: Addison-Wesley. 1973, p. 127.

[2] Robert Berkey, "Integer Division, Rounding and Remainders," *1982 FORML Conference Proceedings,* San Carlos, California: Forth Interest Group, 1983, pp. 13-23.

DDJ

```
( Define 83-Standard /MOD in terms of old /MOD )
 : /MOD   ( num  den  --  mod  quot )
        OVER OVER XOR 0<       ( test signs of arguments )
        IF                     ( signs are different )
             >R R@ /MOD OVER   ( divide and examine remainder )
             IF                ( non-zero remainder )
                  1- SWAP R> + SWAP  ( adjust results )
             ELSE                    ( zero remainder )
                  R> DROP            ( discard old den )
             THEN
        ELSE                         ( signs the same )
             /MOD                    ( just divide normally )
        THEN ;                       ( end of definition )
```

Figure 3.
High Level Forth Code to Convert to Floored Division

When a Page Is Not a Page:
Forth-83 and Vocabularies

FORTH FORTH FORTH FORTH

One of the most frustrating chores in writing a program is not determining the algorithms (that's a challenge) but naming the procedures and program structures so that the program reads and makes sense. Most BASIC programmers don't have this problem because the relationship of the variable name B3 to its usage is just as nonexistent as the relationship of line number 950 to the function of the procedure on that line. When you have 30 or so characters at your disposal, you can pick just about any word names you want. The problem comes in thinking up appropriate, readable names.

Generally, the shortest word possible, so long as the word remains appropriate, will be easiest to use and read. Compound, long, or abbreviated words (especially acronyms) only make the program harder to read and follow. Given that most of us probably did not major in English in college, our working vocabulary is generally limited to an amazingly few thousand. Since most processes are named with appropriately descriptive verbs, the list becomes even smaller.

Vocabularies

The naming problem is solved within the English language by words having context-dependent meanings. Take the word *page*, for instance. Depending on context, *page* can take on any of the following meanings:

> an errand boy or messenger; to summon by calling the name of; a leaf of a book or manuscript or one side of it; to number the pages of, paginate; to thumb through.[1]

Additionally, when referring to memory in computer jargon, a page is some power of two bytes such as 256, 1024, 4096, and so on. It is difficult enough for people to determine what is meant by a sentence (i.e., what do I mean when I say, "The page moved"?) without trying to program the computer to do the same. In fact, on most Forth systems PAGE is defined as performing a form feed (on printers) or clearing the screen (on video terminals).

Forth uses a simple approach to select one of several different meanings for the same word (letter group). Rather than making each word very complicated by trying to have the word determine which meaning was desired, Forth allows the programmer to separate words, even those which have the same names, into (usually) contextual groups referred to as vocabularies. Each vocabulary is simply a list of words that can be selected by name.

How to Use Vocabularies

Vocabularies are simple to create and use. Executing the phrase

```
VOCABULARY FILE
```

creates a word in the dictionary within the compilation vocabulary (the one to which words are being added) named FILE that, when executed, will make a list of words available to the system. Creating the FILE vocabulary does not select the FILE list of words (which is currently empty). Executing the phrase

```
FILE DEFINITIONS
```

selects the FILE list of words to be searched and then tells the system that this is where new word definitions will be placed. Executing the word DEFINITIONS makes the last vocabulary executed (in this case FILE) the compilation vocabulary.

So, to create a vocabulary, make it available, and place word definitions in it, we would execute:

```
VOCABULARY FILE IMMEDIATE
FILE DEFINITIONS
```

The word IMMEDIATE marks the last word created as a word to be executed during compilation. In this case, execution during compilation would select the FILE vocabulary enabling us to access the FILE words. All words defined after executing the above will be placed in the FILE vocabulary until another vocabulary name and the word DEFINITIONS is executed.

Suppose now we wish to define a word to build a file entry into the directory on the disk. Let's see . . . what name should we use to create a file? Think about it. How about CREATE? But there is one small problem. The FORTH vocabulary already has defined the word CREATE to build names into the dictionary. The solution is to use the FILE vocabulary we defined above:

```
FILE DEFINITIONS
  ( we defined FILE above )
: CREATE .... ;
( ". . . ." are your words that
  define how to build a file )
```

Now to create a file we can simply type:

```
.... FILE CREATE ....
  ( ". . . ." are words, or stack or
    other data )
```

Reads nicely, doesn't it? It also saves memory over other naming options such as FILE-CREATE, if other file operation words are going to be defined and the FILE– is going to be repeated.

Search Order

The Forth-83 Standard introduces the concept of search order to Standard Forth systems. The search order is the order in which selected vocabularies in the dictionary are searched. Unfortunately, because the many different ways of implementing vocabularies have varying merits, no Standard has yet been defined for specifying the search order. The current document only tells us:

> The search order begins with the last vocabulary executed and ends with FORTH, unless altered in a system dependent manner.[2]

This acknowledges that Forth systems have the ability to change the search order, but that this cannot be performed in a standard way as yet.

So currently a Forth-83 Standard System searches two vocabularies (at most) during a dictionary search: the vocabulary last executed and then FORTH. Note that this is not a strict rule. The system must simply behave as if this were the case to remain compatible with other systems. What is actually done is up to the implementor; the Standard only specifies the apparent behavior of the system. Also note that the compilation vocabulary (formerly, in Fig-Forth, this was the CURRENT vocabulary) is not necessarily part of the search order as it always is in Fig-Forth.

An Experimental Proposal

All hope is not lost for search order

by George W. Shaw II

George William Shaw II, 24301 Southland Dr., No. 216, Hayward, California 94545.

specification in a Standard System. The Forth Standards have a provision for the proliferation of good but unproven or not widely used ideas that are candidates for future standardization. These are called experimental proposals. One such experimental proposal that has been accepted from William F. Ragsdale is entitled "Search Order Specification and Control" (based on his paper "The ONLY Concept for Vocabularies"[3]). The proposal describes a small set of words for specifying and controlling the search order.

The proposal suggests the addition of five words to the Standard to manipulate and display the search order: ALSO, ONLY, SEAL, ORDER, and WORDS. The first three are used to manipulate the search order and the last two are used for display purposes. Additionally, individual vocabularies are sealed rather than chained as in Fig-Forth and are usually defined as nonIMMEDIATE (depending upon usage).

In Ragsdale's implementation, the search order consists of up to five vocabularies, with the top location being replaced (without affecting the others) each time a vocabulary name is executed. Thus there are up to four permanent positions and one transient position. The vocabulary in the transient position is added to the permanent search order by executing ALSO. The search order is cleared to a minimum set of words (those for vocabulary/search order manipulation only) by executing ONLY. A typical specification of the search order prior to compiling an application might be

```
ONLY FORTH ALSO
APPLICATION DEFINITIONS
```

where FORTH and APPLICATION are previously defined vocabularies. The APPLICATION vocabulary is where the word names about to be compiled will be added.

The word ONLY is an interesting beast; it is actually a vocabulary plus some. Its internal structure is that of a vocabulary, but its execution procedure does what a vocabulary does (placing it first in the search order) and more. ONLY actually removes all vocabularies from the search order and places itself both in the bottommost permanent position and on top. As a vocabulary, ONLY usually contains seven to ten words useful in manipulating vocabularies and the search order: FORTH, DEFINITIONS, VOCABULARY, WORDS, FORGET, ONLY, ALSO, and sometimes SEAL and ORDER. After ONLY is executed the above words are the only words available. In the following, we step through the previous example:

ONLY – clears the search order to minimum words.

FORTH – makes this vocabulary available.

ALSO – keeps the transient item as permanent.

APPLICATION – makes this vocabulary available; it must have been defined previously within FORTH or ONLY vocabularies.

DEFINITIONS – causes new words to become part of APPLICATION vocabulary.

It is very useful to be able to find out what the current search order is, especially when an error occurs because a word name has not been found. The word ORDER performs this job. In TaskFORTH from Shaw Labs (the system I use and the only system being sold as of June 1983 that uses Ragsdale's proposal), the example above would print as:

```
Search:
    APPLICATION  FORTH  ONLY
Compile:
    APPLICATION
```

Also, because this is still experimental, TaskFORTH will hold a search order of up to nine vocabularies. I commonly use six and sometimes seven vocabularies in the search order. I like to use lots of little vocabularies (20 to 100+ words in each) for independent functional areas such as CLOCK (time/date), DOCUMENT (documentation help), DEBUGGER'S (debugging versions of words), and FILE (file system functions) among others.

Another useful search order function is to list the words in the first vocabulary in the search order. WORDS, of course, performs this function. Used by itself, WORDS lists the word names in the last vocabulary executed; to see the list of word names in a given vocabulary, simply execute the vocabulary name first. For instance,

```
EDITOR WORDS
```

will list the word names in the EDITOR vocabulary. The command sequence reads nicely, doesn't it? (The Fig-Forth name for WORDS is VLIST).

The last word name in Ragsdale's experimental proposal is SEAL. This word allows the application to have complete control over the system by removing all programming words from the search order. So, if the word names that should be available to the user are available in the vocabulary COMMANDS, then a command like

```
ONLY COMMANDS SEAL
```

will make COMMANDS the only vocabulary in the search order and hence the only list of words available to the system.

SEAL actually just removes ONLY from the bottom position in the search order where it normally resides. To step through the process in this example, execution of the following commands will produce the corresponding results:

ONLY – allows access to the nine words it contains (COMMANDS should be a part of the ONLY vocabulary).

COMMANDS – makes the COMMANDS vocabulary words available.

SEAL – removes the ONLY vocabulary, leaving the COMMANDS vocabbulary as the sole set of words available.

In practice this can be useful for end-user applications. Since it locks the operator out of the rest of the system, one may thereby prevent any inadvertent or intentional use of words that are unrelated to the application, which could cause problems.

You may have noticed that the experimental proposal doesn't include a word to remove a vocabulary from the permanent search order. This was deliberate on Ragsdale's part. Ragsdale views the search order as somewhat more sacred than a stack, not something that you arbitrarily add to and remove from (other than in the transient position). He believes that if the permanent search order is to be changed you should start over with ONLY. I agree that you should not try to gain the efficiencies of a stack by pushing, popping, and swapping items. But my experience has indicated that I sometimes can simplify my program (for compilation) by temporarily adding a permanent (semi-permanent?) vocabulary to the search order and then removing it. This actually just saves typing or sometimes increases readability.

The word I use for this function is TOSS. It pulls the search order up one slot so that the most recently placed permanent vocabulary moves into the transient position and the previous transient vocabulary is lost. I use this either during compilation (as described above) or from the keyboard to remove a vocabulary that has been ALSOed. My requirement probably stems more from my use of a deeper search order and desire to avoid typing five or six vocabulary names and ALSOs to remove one or more from the top.

Conclusions

Forth-83 Standard is a far cry from Forth-79 in the area of standardized vocabularies. They are more a part of the Standard and seem to be on the leading edge of development. As far as Standard usage is concerned, however, they behave the same way as in Forth-79, which is not saying much. If other vendors were to incorporate Ragsdale's experimental proposal into their Forth systems, a very powerful search order mechanism could by default become a required "nonstandard" part of most Standard Systems.

The Standard seems to allow this through a provision for "Environmental Dependencies," although this provision is directed more towards hardware than Forth system software requirements. If other vendors adopt Ragsdale's proposal, the Forth community could benefit greatly.

References

[1] *The American Heritage Dictionary of the English Language,* 1976, New York: Dell Publishing Company, p. 511.

[2] *Forth-83 Standard,* 1983, Forth Standards Team, P.O. Box 4545, Mountain View, California 94040.

[3] William F. Ragsdale, "The ONLY Concept for Vocabularies," *1982 FORML Proceedings,* pp. 109-115, distributed by the Forth Interest Group, P.O. Box 1105, San Carlos, California 94070.

Additional Reading

William F. Ragsdale, *The Fig-Forth Installation Manual,* distributed by the Forth Interest Group.

George Shaw, Nov. 1981, "Executable Vocabulary Stack," *1981 FORML Proceedings,* vol. 1, pp. 187-194; and Nov. 1982, "Executable Vocabulary Stack – 1 Year of Use," *1982 FORML Proceedings,* pp. 117-130, distributed by Forth Interest Group.

Some Forth Coding Standards

by Edward Wischmeyer

Edward Wischmeyer, Androbotic Inc., 1287 Lawrence Station Rd., Sunnyvale, California 94086.

Forth, like other computer languages, provides a formal syntax for conveying commands to the computer. However, the language will not necessarily convey this information to another programmer with suitable grace. In order to convey this programmer-to-programmer information, coding standards are required to ensure that the information necessary for easy understanding is presented. This article discusses that required information and some different styles of its expression.

What are you doing?

In order to use Forth effectively, you must first ask yourself what you are doing, and what you want to accomplish. Are you trying to kill a half hour before M.A.S.H.? Are you writing game software that has to work, but nobody needs to see inside? Are you writing a Forth system that you wish to sell, and that other people must be able to use and modify? Depending upon what you are doing, you may make certain decisions about how you write your code.

Why coding standards?

Computer programming is, in a sense, nothing more than management of complexity, where ideas are the underlying medium of the complexity, and the ideas are expressed in a code peculiar to the system you are using. The source code tells what you are doing at a very detailed level, which is fine for the computer. People, however, require big-picture information, such as the organization of your code and the requirements imposed by that organization. Conforming to a coding standard should guarantee complete information on this organization, and your choice of complexity management techniques, regardless of the format used.

Coding standards describe various aspects of code generation, including structure, modularization, names of words (procedures), comments, and layout of typed source code. Structure and modularization are the same in Forth as in Pascal, C, Ada, and other structured languages, and have been dealt with at length in standard texts. This article discusses aspects more peculiar to the Forth environment, including names, comments and layout.

Names of Forth Words

English is a powerful, subtle, complicated, sophisticated, yet ambiguous language. For a native speaker of English, Forth provides a unique facility for expressing ideas to a computer. Since a Forth word is invoked (either for compilation or execution) by typing its name, the name is a logical place to start looking at Forth coding standards.

Long vs. Short

A major bone of contention is whether one should use long word names or short word names. The common-sense answer is, it depends. For example, if you are controlling a color display, you might choose to have words with names like

```
RED   BLUE   GREEN   BLACK
   WHITE   SET_FOREGROUND
      SET_BACKGROUND
```

Another choice would be to have words like

```
RED_BACKGROUND
    RED_FOREGROUND
BLUE_BACKGROUND
    BLUE_FOREGROUND
GREEN_BACKGROUND
    GREEN_FOREGROUND
BLACK_BACKGROUND
    BLACK_FOREGROUND
WHITE_BACKGROUND
    WHITE_FOREGROUND
```

or perhaps

```
RED          BACKGROUND      SET
BLUE         FOREGROUND
GREEN
BLACK
WHITE
```

The first implementation seems to have several advantages: the color words and the SET_ words allow you to choose your own combinations, so that additions to the color set require no knowledge of changes in the SET_ commands, and changes to the SET_ commands do not affect the color words. By comparison, if you had to change, say, the _BACKGROUND part in the second set of words, you would have five words to change, rather than just one. The total length of all commands is shorter, which may not be important. The third choice may be appropriate in some situations.

The way to tell which alternative to use is to examine the underlying parallelism (if any) of the internals. If your software consists of various parallel modules with very similar structure, then it makes sense to use short words with parallel meanings, such as RED, BLUE, GREEN as one set, and FOREGROUND and BACKGROUND as another set.

Often it is appropriate to use longer word names, particularly for words which both cause something to happen (as opposed to those related to memory structures) and are specialized to the point that no other Forth words in your system will perform the same function. Some examples would be

```
RESET_LOG_FILE
TRANSMIT_DATA_PACKET
```

These longer word names are usually found in applications software (as opposed to systems software), and in the higher levels of the software – the parts that call other routines to do the smaller tasks which make up the whole.

Consider another set of words which control a faucet:

```
FAUCET          OPEN        CLOSE
```

versus

```
FAUCET_OPEN     FAUCET_CLOSE
      or
OPEN_FAUCET     CLOSE_FAUCET
```

Which makes more sense? Possibly, the first one makes more sense if FAUCET specifies what is to be affected by the command, and the two parallel commands OPEN and CLOSE operate the specified device. If, however, OPEN and CLOSE can work only on FAUCET, it makes sense to consolidate the words into OPEN-FAUCET and CLOSE-FAUCET.

Word Combinations

For those situations in which long names are appropriate, a good idea is to start the long name with a verb. In the example above, SET_BACKGROUND starts with a verb. So does OPEN_FAUCET – or does it? Does OPEN_FAUCET mean "open the faucet" or does it mean "a faucet which is open"? Here is where the ambiguity of English can make things interesting. The same

language that supports an especially wide range of slang expressions and that is rich enough to provide enough words to be the world's *de facto* technical language, has enough loopholes to let you write confusing gibberish. After all, it may not, for your application, be enough that you understand what you are writing now, or that one of the possible ambiguous meanings makes sense. If you need to convey ideas, ambiguities can do you in.

There is, not surprisingly, another opinion. That opinion is that the verb should *end* the name, just as Forth operators come after their arguments in postfix notation. For example, you might write FAUCET_OPEN, or, to make it more English-like and less Forth-ish, FAUCET_OPENS or FAUCET_IS_OPENED.

In my experience, verb phrases, such as SET_BACKGROUND, make excellent names for Forth words that cause something to happen. For Forth words that are primarily memory access, and which generate only addresses or values, noun phrases and nouns are appropriate, such as BUFFER_LENGTH or BYTES/BLOCK.

By this time, you have probably noted that much of this discussion is about writing, and not particularly about Forth. If you are concerned with transmitting ideas, writing skills are as important in Forth as technical skills.[1]

Abbreviations

One argument that comes up from time to time in the long/short name controversy is the time spent typing. There is no doubt that long names will take longer to type, and they will probably take up more space, too. Often, though, this is only part of the issue. If there is any debugging to be done, or if the code must survive over time or be accessed by multiple programmers, then the reading and understanding time can be much more important than the typing time. On the other hand, if you have words which are used frequently enough that contractions can be remembered easily, then these contractions may be appropriate. After all, the Forth words

```
DUP     ROT
```

are widely used and accepted, rather than the more formal

```
DUPLICATE_TOP_OF_STACK
ROTATE_TOP_3_ON_STACK
```

There is a pitfall in using abbreviations, however. If you use different abbreviations for the same English word in different Forth words, or sometimes abbreviate a word and sometimes not, you will probably go bonkers trying to keep the different abbreviations straight. Consider, for example

```
ARDVK_PRICE      ARDVRK_AGE
AARD_WEIGHT      RDVRK_LENGTH
```

Four different abbreviations for aardvark, and which abbreviation goes with what? The moral of this story is that anything abbreviated once must always be abbreviated, and always in the same way.

Almost every programmer optimizes his code. Some optimize for speed, some for minimum memory usage, some for clarity of source code. If you optimize your code for minimum number of typed keystrokes, make sure that you can afford a price which is often high and seldom predictable.

Commenting Your Code

If you ever want to rile your acquaintances who have heard of Forth but don't use it, tell them that it is self-commenting (i.e., the code is so clear that no comments are required). Then show them some code that looks like

```
: EUREKA!!! OVER >R DP HERE
    SWAP ! SWAP DROP R> SWAP ;
```

On the other hand, you could show them code such as

```
: INITIALIZE_SCREEN
    CLEAR_SCREEN
    SET_UPPER_&_LOWER_CASE
    ENABLE_COLOR_OUTPUT ;
```

which gives a reasonable approximation to self-documentation.

In both of these examples, however, there is a shortcoming. If you need to use EUREKA!! or INITIALIZE_SCREEN, what inputs do you provide? What outputs can you expect? What formats do these inputs and outputs use? In Forth, words are really procedures that cause some sequence of steps to execute. Many of these procedures are functions in the mathematical sense, with a domain (set of inputs) and a range (set of outputs). If you are to specify this function completely, you need to give the inputs required, and the outputs generated. If this information is not provided, you have generated the Forth equivalent of a graph with no labels on the axes. Maybe you knew once what the graph meant, but how can anybody else figure it out? Will you remember it?

To solve the input/output commenting problem, Fig (the Forth Interest Group) developed a set of stack comments which were appropriate for their needs – specifically, documenting the inputs and outputs to the basic kernel words. Briefly, the Fig stack comments take the form

```
( a b --- c )
```

which means that a and b are inputs, with b (the right-most item) on top of the stack, and c as the output. In the Fig documentation, the types (integer, double integer, byte, etc.) of the inputs and outputs are specified, rather than the names of the inputs and outputs. (See the *Fig-Forth Installation Manual.*[2]) This is appropriate when your Forth word operates on any 16-bit address, or on any true/false flag, or on any byte.

When you write applications software, however, you are attempting to solve a different set of problems, and the stack comments appropriate to that different set of problems may also be different. For example, if you have a complicated Forth word with four inputs and two outputs, commenting that these inputs and outputs are all 16-bit integers will probably not be very useful. For applications, you will undoubtedly want to specify the names of the inputs and outputs. If your names are multi-word, such as LEFT_MOTOR_RPM, you may put dashes (-) or underbars (_) between the elements of your long input and output names, and separate these long names from each other with spaces. If you use multi-word names and use spaces to separate the elements of each multi-word name, you will have to separate the names with something more obvious, such as commas. Additionally, you may have some words that access information on the stack but do not consume it. (Some parts of the stack may be read but not changed.)

On a past Forth project with three programmers and a $250,000 budget, we developed our own stack comments to fulfill our requirements of readability and modifiability of the code by knowledgable programmers. We wrote and published our own stack comments, confident that anyone reading our code would be able to decipher them. Our stack comments took the following form

```
: CALC_RADIATED_POWER
    (  unit type; mode --> eff rad
       power in dBm)
```

The translation scheme was that everything to the left of the semicolon was unchanged, but required as an input. Things to the right of the semicolon were changing parts. As you might expect, things to the left of the arrow were consumed, and things to the right of the arrow were generated. The semicolon was always required, the arrow was required only when the Forth word generated an output. As with the Fig comments, items are listed in the order they appear on the stack, with the least accessible items in the left-most position. Thus, things that are not consumed must be placed on the stack before items that are consumed. The Fig equivalent of the above stack comment, with the understanding that repeated items are unchanged, would be

```
( unit_type mode --- unit_type
    eff_rad_power_in_dBm )
```

We developed our scheme, rather than using the Fig scheme, because we needed to keep track of both changed

and unchanged variables on the stack, which the Fig scheme did not explicitly support. Both schemes, however, conveyed the fundamental information that inputs are required and outputs are generated. The disadvantage was that we had two different stack comment schemes. Our rather complicated applications used one scheme, and our Fig-Forth kernel used a different scheme. For our purposes, the two different schemes worked fine, but your purposes will dictate your choice.

Yet another way of commenting your code keeps track of the paramenter stack, the return stack, textual input, memory, and whatever else. In this scheme, each of these data sources/sinks has its own comment line, with a prefix on each line indicating to what the comment applies. For example, you might see

```
( PS    disk drive - - - )
( KB    file name - - - )
( M     buffers - - - buffers )
```

This notation would indicate that the parameter stack input was a disk drive number, that the required keyboard input would be a file name, and that the disk buffers would be modified.

As a final embellishment, each commented item could include an identifier relating to the format of the data. Using the example above, but with i standing for integer and b standing for byte,

```
: CALC_RADIATED_POWER
    ( b unit type; b mode  ->
      i eff rad power )
```

This gives really complete information about inputs and outputs, but may be complicated to decipher if you have your own data types.

The whole purpose of stack comments is to communicate the inputs and outputs of your Forth words. For your applications and your code readers, there will be good ways and bad ways of communicating. Forth does not force you into any set pattern, making you responsible for choosing your own methodology.

Other Forth source code comments are typically three types: what you are doing, how you are doing it, and what is on the stack at the end of each line. Most of the time, the name of the Forth word tells what you are doing. In idealized, hypothetical Forth programming, you never have to give a comment concerning how you are doing this function because you connect the words in your comment with underbars of dashes and your comment becomes an executable Forth word. This is high-level coding at its best, as exemplified by INITIALIZE_SCREEN above. The price of this, however, is that you may have a lot of Forth words that are only called once.

At some point you will not be using high-level words, but will have to revert to the low-level Forth kernel words. (This is the same phenomenon which dictates that, when writing a Forth kernel, there comes a point at which you have to revert to the underlying assembly code of the machine.) When you get to this lower level, chances are that there will be a fair amount of stack manipulation. Typically, 30% of your code will be stack manipulation words, such as SWAP, DROP, DUP, and the rest. Here it may be appropriate to put a comment at the end of each line so that you can keep track of what is on the stack, particularly in long or tedious calculations.

Spacing Forth Words on a Line

A good rule of thumb in spacing Forth words on a line is to group the words into sub-phrases, phrases, and longer groups, just as English is punctuated in phrases and clauses. Most Forth people agree that closely related words should be separated with single spaces into short sub-phrases, and these sub-phrases should be separated by multiple spaces. Consider the example in Figure 1 (page 100).

In the definition of ABC, notice that the ARRAY-SIZE and the 2- are clustered into one sub-phrase. In the definition of DOUBLE_THE_ARRAY, again note the obvious phrasing. None of this is subtle or difficult, but it does make things more readable than an equivalent:

```
: 2*IT 0 DO DUP I + DUP @ DUP
    + SWAP ! 2 +LOOP DROP ;
```

To really clean this up, you could type the code as shown in Figure 2 (page 100). Although this simple example really does not need all of these comments, the time spent typing them in is minimal. The real effort of Forth programming is figuring out what you want to do and how to do it, much of software development time is figuring out what you actually did. A short time spent typing can preserve a lot of thinking and figuring.

Finally, consider control constructs, such as IF ELSE THEN and DO LOOP. Most people consider it good practice that any control construct which begins a block of code starts a new line. The block of code underneath is usually indented. DOUBLE_THE_ARRAY above shows a DO phrase starting a line, and the block of code underneath it is indented. In addition to the layout of words on a line, it is important to clearly structure your code across lines. It seems obvious that each short sub-phrase should be intact on a line. It is also traditional to indent to show the blocks which together make up a word, as shown in Figure 3 (page 101). Although many people will argue whether a THEN should be at the end of a line or at the beginning of a new line, or whether DO should start a line, there is agreement that the underlying block structure of a Forth word should be reflected in the layout of the code on the screen.

Miscellaneous Remarks

There are other forms of coding standards that you can use in your system. For example, on the large project mentioned above, we had a standard that all words ending with a question mark had to leave a true/false flag on the stack. This gave us more readable code, such as

```
IS_THIS_REALLY_THE_CASE?
    IF DO_SOMETHING_ABOUT_IT
    ELSE DO_NOTHING THEN
        THEN
```

Another useful idea was that all of our words which began with *GET_ would* add to the stack. This was a tremendous simplification. A final technique we employed was to separate our high-level code and low-level code. High-level code was comprised of user-defined words and Forth control constructs. Low-level words were Forth kernel words and user-defined memory constructs such as variables, constants, and arrays. Our goal was for all of our user words to be either high level or low level, but not to incorporate elements of both high and low level into the same word. This meant that we could show our non-software bosses our high-level code, with no comments, and they could understand it. The low-level code, with all the stack comments, we kept to ourselves. Our success was demonstrated by our ability to debug each other's code without having seen it before and having no supporting documentation, and by the number of times major modifications to different modules worked together on first integration.

Conclusion

Forth provides great powers to the programmer to command the computer, but these powers do not guarantee easy communication from programmer to another human being. By adhering to a coding standard, you can ensure that your code contains information that other people (not machines) need to see in order to more easily understand your code. In Forth, this information *must* include: stack comments to specify inputs and outputs to each word; appropriate names for words to convey the organization and function of your code; and appropriate layout of source code on the page. The variety of purposes of the applications and the programmers, plus human nature, will conspire to insure that no one format will be universally accepted, but common sense will dictate a minimum amount of information that must always be included.

References

[1] Strunk and White, 1979, *The Elements of Style,* McMillan Publishing Co., New York, New York.

[2] William F. Ragsdale, 1980, *Fig-Forth Installation Manual*, Forth Interest Group, P.O. Box 1105, San Carlos, CA, p. 7.

DDJ

```
20 CONSTANT ARRAY-SIZE
VARIABLE ABC   ARRAY-SIZE 2-   ALLOT  ( 20 BYTES TOTAL RESERVED )

: DOUBLE_THE_ARRAY          ( ;ARRAY ADDRESS, ARRAY SIZE IN BYTES )
   0 DO   DUP I +   DUP @ DUP +   SWAP !   2 +LOOP   DROP ;
```

Figure 1.

```
: DOUBLE_THE_ARRAY       ( ;array address, address size in bytes )
   0 DO
     DUP I +            ( ;array address, address of Ith element )
     DUP @ DUP +         ( ;array addr, Ith addr, 2* Ith element )
     SWAP !   2 +LOOP    DROP ;
```

Figure 2.

```
: UNDO ( ; )
    UNDOABLE?
      IF COMPILING?
        IF EXECUTING?
          IF   UNDO-IN-DEF&DO
          ELSE UNDO-IN-DEFINE THEN
        ELSE EXECUTING?
          IF UNDO-EXECUTE NOTUNDOABLE THEN THEN
      ELSE CANTUNDO THEN ;
DECIMAL EXIT
```

Figure 3.

The Forth Sort

The rapid sorting of data is probably one of the most useful and most used capabilities of computers, while at the same time being one of the least understood. There are astonishing differences in the performance obtained with various sorting algorithms under different conditions, and to optimize sorting efficiency requires knowledge of how certain factors interact in any given sorting environment.

As an aid to the understanding of computer sorting, this article will first explain three relatively simple algorithms, and then, on the theory that "one picture is worth a thousand words," will present a user friendly sorting demonstration program, which graphically illustrates differences in performance. The program is coded in Forth because its inherently structured form helps with visualizing and understanding the action of the sorting algorithms.

The first and simplest of the algorithms to be examined is the well known Bubblesort. This algorithm starts by comparing the second item with the first item in the list to be sorted. If they are not in correct relative order, it swaps them. It then goes on to compare the third item to the first, then the fourth to the first and so on, until all items have been compared to the first and swapped as required. When this initial sequence is complete, the first item in the list, and only the first, is guaranteed to be in the correct position.

Bubblesort then continues with another sequence in which all items further down the list are compared to the second item, swapping as necessary. When this sequence is complete, the first two items in the list will be in correct order. The process continues for number of items in the list minus one sequences (n-1) of advancing one item and comparing all items further down the list to it. As can be seen, each sequence involves making one less comparison than the previous sequence, and the final sequence involves *only the last* two items. At this time the whole list is in order and the process ends.

by Mark Manning

Mark I. Manning, 7611 Autumnal Lane, Liverpool, New York 13088.

Bubblesort has the distinction of having the lowest performance rating of the three algorithms being examined, because it must always perform n-1 of the above described sequences and, for each sequence, must always compare the selected item with all of the remaining items in the list. Thus when the number of items to sort is doubled, the sorting process takes approximately four times longer, reaching unacceptable limits very quickly.

The second algorithm to be examined is the less well known Shuttlesort, which is in the same efficiency class as Bubblesort but which has a refinement that provides a surprising performance difference. Shuttlesort begins by comparing the second item to the first, swapping them if necessary. It then proceeds to compare the third item to the second, then the fourth to the third and so on, until it reaches the end of the list.

When a pair of items being compared are in the wrong order, they are swapped and Shuttlesort backs up to compare the item just swapped with the previous item in the list, swapping again if necessary. The backing up process continues until a pair in the correct order is found, at which time Shuttlesort jumps forward to a point one item ahead of where the backing up started and continues where it left off.

The described action causes out of order items to "shuttle," one at a time, from their initial position to their final position, and is also the basis for the above mentioned surprise performance. If Shuttlesort is run on an already-ordered list, it will take less time to complete its task than any other algorithm in existence, regardless of the size of the list. It is also quite fast on nearly ordered lists.

The final algorithm to be examined is the Shell-Metzner Sort, or Shellsort as it is commonly called. Shellsort is basically the Shuttlesort algorithm, with a small, but all important, additional feature that increases its efficiency considerably. With Shellsort, doubling the number of items to be sorted results in only a 2.3 to 2.4 fold increase in sort time, providing an excellent tradeoff between algorithm complexity and efficiency.

The Shellsort algorithm first sets an initial distance, which is something more than half the total number of items in the list to be sorted. It then compares items in the list, separated from each other by this initial distance, using the Shuttlesort method of stepping down the list making comparisons and swaps. When it completes the list using the initial distance, it decreases the distance by half and runs down the list again. The process of decreasing the distance by half and making a Shuttlesort pass down the list continues until the final pass is made using a distance of one.

This final pass is identical to a Shuttlesort, but is very fast, since the previous passes have left the list in near sorted order. Shellsort owes its speed and efficiency to the fact that because it can move out of place items over initially large, and then successively finer distances, it makes fewer comparisons and swaps than the other algorithms discussed. If run on an already ordered list, however, it does not show as dramatic a decrease in run time as Shuttlesort, because it must make a separate pass for each distance.

The demonstration program will provide an interesting side-by-side comparison of the three sorting algorithms discussed, under three different conditions. The first sort is made on randomly ordered data, which represents an approximation of a typical sorting situation. Next the same algorithm is run a second time on the sorted data, to observe the algorithm's sensitivity to the data order by contrasting best case conditions. Finally, the algorithm is run again on data in reversed chronological order, to observe the algorithm's sensitivity to worst case conditions.

The demonstration program (written in Fig-Forth 79 2.1) is straightforward and should run on all dialects of Forth with only small changes. The following comments will help resolve any system differences:

1. Fig-Forth users should change NEGATE in SHELLSORT to MINUS.
2. 79 Standard Forth users should change the -2 loop limit for the inner DO loop of both SHUTTLESORT and SHELLSORT to 0.
3. BEEP is defined as 07 EMIT.
4. PAGE is defined as 0C EMIT.
5. The CONSTANT ARRAY1 defines the starting address for the array to be sorted. The array address was allocated directly, in order to allow arbitrarily large arrays to be created, without having to ALLOT a large block of Forth's dictionary space and to demonstrate the flexibility of Forth. Those users

who do not have sufficient room outside of Forth's program space or who are uncomfortable with this method may use the more traditional procedure of:

1. CREATE ARRAY1 XXX ALLOT – for 79 Standard.
2. 0 VARIABLE ARRAY1 XXX-2 ALLOT – for Fig Forth.

References

Donald Knuth, *The Art of Computer Programming,* Vol. 2, (Sorting and Searching), 1973, Menlo Park: Addison-Wesley.

Harold Lorin, *Sorting & Sort Systems,* 1975, Menlo Park: Addison-Wesley.

►►J

The Forth Sort (Text begins on page 102)

```
SCR # 21
  0 ( 16 bit Numerical Sort Demo                          20AUG82MIM)
  1
  2 FORTH DEFINITIONS  HEX
  3 CREATE ARRAY1 402 ALLOT    ( address of data array)
  4 VARIABLE #ELEMENTS      ( number of 16 bit elements)
  5 VARIABLE DISTANCE       ( distance between elements)
  6 VARIABLE VI             ( temporary indexes for nested "DO"s)
  7 VARIABLE VJ
  8 VARIABLE SEED   HERE SEED !
  9
 10 : RND            ( random # generator)   ( n --- )
 11    SEED @ 103 * 3 + 7FFF AND
 12       DUP SEED !  7FFF */ ;
 13 : CLRS PAGE CR CR 17 SPACES ." FORTH SORTING DEMO" CR ;
 14 : KEYMSG  ." any key continues.." CR KEY DROP ;
 15


SCR # 22
  0 ( 16 bit Numerical Sort Demo                          20AUG82MIM)
  1
  2 : RANDOM                    ( create random pattern in ARRAY1)
  3    #ELEMENTS @ 2* 0       ( set loop limit and initial index)
  4       DO 3E8 RND          ( fetch random # between 0 and 999)
  5          I 3 MOD 0= IF NEGATE      ( negate 1 out of three)
  6       THEN I ARRAY1 + !               ( store in array)
  7          2 +LOOP ;                    ( increment loop)
  8
  9 : REVERSE                ( create reversed pattern in ARRAY1)
 10    #ELEMENTS @ 0          ( set loop limit and initial index)
 11       DO #ELEMENTS @ I -               ( compute value)
 12          I 2* ARRAY1 + !              ( store in array)
 13       LOOP ;                          ( decrement loop)
 14
```

(Continued on page 106)

```
SCR # 23
  0 ( 16 bit Numerical Sort Demo                          20AUG82MIM)
  1
  2 : NUM(I) @ ARRAY1 + ;                ( array fetch)
  3 : NUMI@ NUM(I) @ ;                   ( and store)
  4 : NUMI! NUM(I) ! ;                   ( operators)
  5
  6 : COMPARE VI NUMI@ VJ NUMI@ > ;     ( true if #I > #J)
  7
  8 : NUMSWAP           ( swap elements of array)   ( --- )
  9    VI NUMI@ VJ NUMI@ VI NUMI! VJ NUMI! ;
 10
 11 : NUMLIST               ( output number array)   ( --- )
 12    #ELEMENTS @ 2* 0 DO I DUP
 13       1A MOD 0= IF CR THEN ARRAY1 + @
 14          6 .R 2 +LOOP CR CR ;
 15


SCR # 24
  0 ( 16 bit Numerical Sort Demo                          20AUG82MIM)
  1
  2 : BUBBLESORT                ( sort data array)   ( --- )
  3    #ELEMENTS @ 1- 2* 0 DO I VI !
  4       #ELEMENTS @ 2* I 2+ DO I VJ !
  5          COMPARE IF NUMSWAP
  6       THEN 2 +LOOP 2 +LOOP ;
  7
  8 : SHUTTLESORT               ( sort data array)   ( --- )
  9    #ELEMENTS @ 1- 2* 0 DO
 10       -2 I DO I DUP VI ! 2+ VJ !
 11          COMPARE IF NUMSWAP ELSE LEAVE
 12       THEN -2 +LOOP 2 +LOOP ;
 13
 14 ( For decending sorts change > in COMPARE to <)
 15


SCR # 25
  0 ( 16 bit Numerical Sort Demo                          20AUG82MIM)
  1
  2 : SETDIST                ( set initial distance)    ( --- )
  3     1 BEGIN 2* DUP #ELEMENTS @ >
  4          UNTIL 2- DISTANCE ! ;
  5
  6 : DECDIST                ( decrement distance)   ( --- flag)
  7     DISTANCE @ 2/ 2/ 2* DUP DISTANCE ! 2 < ;
  8
  9     ( Shell-Metzner sort)
 10 : SHELLSORT  SETDIST BEGIN     ( sort data array) ( --- )
 11    #ELEMENTS @ 2* DISTANCE @ - 0 DO -2 I DO
```

```
 12          I DUP VI ! DISTANCE @ + VJ ! COMPARE IF
 13             NUMSWAP ELSE LEAVE THEN DISTANCE @ NEGATE
 14                +LOOP 2 +LOOP DECDIST UNTIL ;
 15


SCR # 26
  0 ( 16 bit Numerical Sort Demo                          20AUG82MIM)
  1     ( benchmark it)
  2 : #ELEMENTS? CR ." How many elements?  " QUERY CR CR
  3     INTERPRET #ELEMENTS ! ." random array" RANDOM NUMLIST ;
  4 : REVIT CR ." reversed array" REVERSE NUMLIST ;
  5
  6 : BUBBS #ELEMENTS? ." random bubblesort.." CR BEEP
  7     BUBBLESORT BEEP NUMLIST KEYMSG ." sorting sorted array.."
  8     CR BEEP BUBBLESORT BEEP KEYMSG REVIT
  9     ." reverse bubblesort.." CR BEEP BUBBLESORT BEEP
 10     NUMLIST KEYMSG ;
 11 : SHUTS #ELEMENTS? ." random shuttlesort.." CR BEEP
 12     SHUTTLESORT BEEP NUMLIST KEYMSG ." sorting sorted array.."
 13     CR BEEP SHUTTLESORT BEEP KEYMSG REVIT
 14     ." reverse shuttlesort.." CR BEEP SHUTTLESORT BEEP
 15     NUMLIST KEYMSG ;


SCR # 27
  0 ( 16 bit Numerical Sort Demo                          20AUG82MIM)
  1
  2 : SHELS #ELEMENTS? ." random shellsort.." CR BEEP
  3     SHELLSORT BEEP NUMLIST KEYMSG ." sorting sorted array.."
  4     CR BEEP SHELLSORT BEEP KEYMSG REVIT
  5     ." reverse shellsort.." BEEP SHELLSORT BEEP
  6     NUMLIST KEYMSG ;
  7 : DECODE DUP 31 = IF BUBBS ELSE DUP 32 = IF SHUTS ELSE
  8     DUP 33 = IF SHELS ELSE 34 = IF QUIT THEN THEN THEN THEN ;
  9 : MENU CR CR ." Specify sort algorithm:" CR
 10     ." 1 - Bubblesort" CR ." 2 - Shuttlesort" CR
 11        ." 3 - Shellsort" CR ." 4 - Exit demo " BEGIN
 12     KEY DUP 30 > OVER 35 < AND NOT WHILE DROP REPEAT ;
 13
 14 : DEMO BEGIN CLRS MENU DUP DECODE AGAIN ;
 15         DECIMAL
```

End Listing

CP/M EXCHANGE

by Robert Blum

Some of the most sensitive and difficult to understand parts of BIOS code are the sections dealing with disk control. Reviewing what must be accomplished and the associated problems is prerequisite to digging in and making changes. Since it seems that writing a new BIOS for CP/M Plus is practically mandatory, a brief review of disk controllers and recording methods is in order.

FM and MFM Recording

When recording data on magnetic media the single most limiting factor to how much data can be stored is the media itself. To record one bit of data requires that some magnetic indication be given that a state change has taken place. In its simplest form this change is a flux reversal. This can be viewed as driving along a newly paved road. It feels perfectly smooth to you only because the suspension system of your car has been designed to filter out any minor irregularities in the road's surface. But when coming to an intersection guarded by speed strips the suspension system's ability to filter out bumps is exceeded, making you aware that something has changed. Magnetic media also has a background or residual noise level present. It is of little consequence because it is filtered out electronically. As the speed strip got your attention so does a data bit as it passes under the read/write head of your floppy disk drive. Its presence is known by a voltage potential well above the normal background noise level.

When designing a magnetic recording and recovery system, a great number of trade-offs must be made. A suitable cost/performance ratio for small computers is met by maintaining relatively low recording densities that allow lower priced media and fairly large mechanical variances to still give acceptable levels of performance.

For example, let's assume that one bit cell time (the minimum amount of time necessary to safely achieve one flux *reversal) is* 2 microseconds (millionths of a second) on a disk rotating at 360 revolutions per minute (RPM). A quick calculation reveals that over 83 thousand bits can be recorded in a single revolution of the disk. Unfortunately, not all of this recording area can be used for data storage. Some of it is used for overhead functions such as identifying sectors and storing the checksums used for data verification on read-back.

Frequency Modulation (FM), also known as single-density recording, was the first to emerge and still retains its position as the standard interchange format for 8-inch disk drives. Because of its low recording density it is seldom used today with 5-inch drives. For example, my first disk system was a North Star MDS that used Shugart SA400 5-inch, single-sided, single-density drives. Maximum storage capacity of each disk was limited to only 90K bytes. At that time nothing else was available that provided more storage, that would also fit into my budget, so a great deal of disk swapping went on.

FM recording, Figure 1 (page 112), offers the lowest recording density because each data bit is encoded with one clock bit. Very reliable data retrieval, even within a wide mechanical tolerance specification, is insured by using this method; however, total data capacity suffers.

To achieve greater storage capacity without significantly increasing hardware and media costs was the mother of invention to Modified Frequency Modulation (MFM) recording, Figure 2 (page 112). This method does not require accompanying each data bit with a clock bit. Clock bits are only inserted when two consecutive bit cells contain zero data bits. This insures that the longest delay between any two flux reversals will be one bit cell time. When MFM was first introduced it was met with a great deal of skepticism. At that time many computer manufacturers made their thoughts perfectly clear: double density would at best be barely capable of maintaining the excellent error rates of FM recording. Fortunately for us, the design engineers knew better and continued to refine read/write heads and their controlling electronics. Today double density is the standard recording method.

Making It Go Click . . . Whir

My disk controller board is built around the very popular Western Digital 1793 that is used in many of today's computer systems. The WD1793 is only one in a large family of disk controller chips that evolved several years ago to meet the demand for floppy disk based systems. Prior to their introduction most disk controller subsystems were built from TTL parts. This required that a large portion of PC board real estate be devoted to just this one function. In a bus-oriented system, such as the S-100, space isn't a major problem because plenty of extra slots are generally available. However, as desktop computers became more popular, devoting a large amount of PC board space to just one function came under close scrutiny of the design engineers. Additionally, heat dissipation and cost were also factors that demanded a more compact approach to controlling floppy disks.

Western Digital's first single-chip controller that gained widespread acceptance was the 1771. It was designed to control either 5- or 8-inch single-density single-sided disk drives. As disk drive manufacturers responded to the need for more storage by introducing drives with dual heads and double-density recording capabilities WD introduced the 179X series. To build a complete disk controller using one of these chips only requires a small amount of external logic to handle data separation and bus control. Further reductions in the amount of supporting logic has been brought about by WD's recent announcement of the WD279X series that incorporates the data separator logic on chip.

Contained on each of these single chip controllers is a complete logic subsystem rivaling the power of most 8-bit cpus in use today. After receiving a few parameters and a single command they will independently perform all the functions necessary to properly position the drive R/W heads and transfer the desired data sector to or from the recording media. For example, to read a sector requires only four software steps. First, load the track register with the target track number. Second, place the desired sector number into the sector register. Third, load the command register with a SEEK operation code. Upon receipt of the SEEK command the controller chip automatically steps to the desired track and verifies the final head position by reading the first sector id that comes along. If the disk heads were not loaded at initiation of the seek they are inserted with the appropriate head settle delay to insure reliable operation. Providing no error is found during the seek operation we are ready for the final step. Finally, load the command register with a READ command. This sets the controller chip into scan mode which reads each sector id until one is found that matches the value placed into the sector register during setup. The data portion of the sector is now transferred and the data content

CRC is checked to verify that the read operation was successful. Obviously, this is a simplified description of a complicated task performed by the controller chip, but from a software viewpoint little else is required.

If you want to know more about any of Western Digital's controller chips you can contact them directly at: Technical Publications, Western Digital Corporation, 3128 Redhill Avenue, Box 2180, Newport Beach, California 92663. They have a number of technical data sheets and application notes available at reasonable cost.

Catch That Error

Preparing to write a new BIOS for CP/M Plus has been very revealing of my hidden work habits. Soon after starting review of my current BIOS I was struck by a flood of "Things To Do" that I had never completed. Most of them were convenience items, such as allowing a drive not ready condition to be cleared with a control-C and adding improved I/O redirection capabilities. But some others could not possibly be dismissed as wish list items.

One problem that I face daily is using a disk system that is beginning to show its age. Far too many hours of constant use without any preventive maintenance has taken its toll. This fact in conjunction with the rather unsophisticated error recovery routines of my BIOS causes me consternation each time I start a backup of my not easily replaced disk files. This is particularly true when I attempt to copy a complete disk. I say attempt with no pun intended. More often than not I encounter at least one file that will not verify properly because of a read or write error. Usually it will copy successfully on the second or third attempt; however, I have been greeted with disaster several times when multiple recovery attempts failed and I had to recreate the file from scratch. Determining exactly what is causing the problem on my system is not nearly as simple as it could be. I use boards from a number of different manufacturers. All are plugged into a motherboard that was never designed to handle the bandwidth in use. And the edge connectors have aged beyond being able to provide a tight connection. Obviously, the easiest solution to my dilemma would be to upgrade my equipment and send the disk drives out for a thorough cleaning and recalibration. But, the buffalo on the nickel bellowing and my sheer fright at being without a computer for several weeks have intimidated me to the point of waiting and hoping for the best. I understand all too well that one day soon I may be faced with a pile of smoking rubble that was once my beloved friend and companion on many dark nights and rainy afternoons. But for now, hope continues to mask the stark realities.

To shorten the length of down time I have enhanced the error reporting logic of my BIOS in hopes of being able to pinpoint the problem. My original intent was to put in enough code to clearly explain in English that an error had occurred and exactly what it was. I also wanted to keep track of each error by type so that totals could be displayed at warm boot time. What I actually implemented leaves a lot to be desired because of the limited amount of space remaining in my BIOS, but it serves its purpose. The routine I wrote (Listing One, page 114), will display, in binary, the status of the disk controller after 10 recovery attempts have failed to clear the error. With this information it is then a simple task to find the error in the manufacturer's literature.

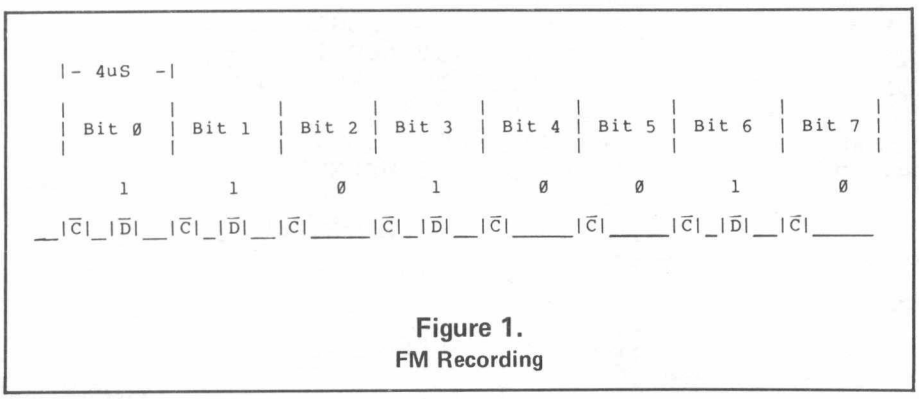


Figure 1.
FM Recording

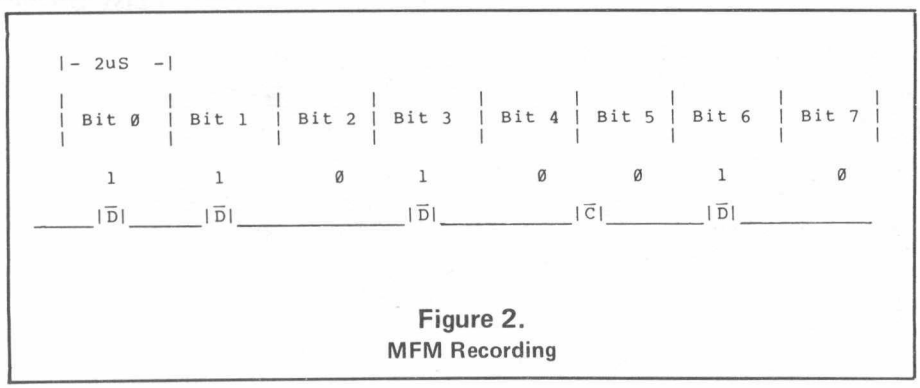


Figure 2.
MFM Recording

TWITTLE

Sitting in front of my CRT for several minutes wondering whether my program has gone south or is just taking its time led me to write a subroutine to visually give some indication as to what is going on. I didn't want to slow a program's execution considerably by displaying long messages or writing a lot of extra code for testing that would be stripped out later. TWITTLE, (Listing Two, page 116), meets both of these requirements for me.

TWITTLE displays an asterisk in the current cursor position that appears to bounce as the program runs. Careful selection of the points in your program where TWITTLE is called will give you not only visual indication that your program is still active but its level of activity as well.

TWITTLE works by using a one-byte save area as a flip-flop flag. On each entry this flag is tested to determine what was displayed on the last entry. In this way the subroutine knows to display either an asterisk or a space to turn the asterisk off. To avoid destroying any data on the CRT screen by outputting a long string, a backspace is output after each indicator character.

As I said before, be mindful of where TWITTLE is called. Calling it too often will produce what appears to be a steady asterisk while not calling it often enough still has you sitting and waiting for something to happen. As a guide, I used TWITTLE in a program that read in a disk file, selecting certain records and then sorting them. I called TWITTLE before reading each input record and then again as each sort pass completed. While reading the disk file the frequency of the bouncing asterisk indicated how many records were being selected. And as the table was being sorted a steady indication assured me that the program was still active.

I deviated from my charted course over the past couple of months based on your input. Next month I plan to cover BDOS function code 31 in greater detail and talk some more about CP/M Plus's features. I also hope to be able to tell you about a new program that was offered to me. It looks like a super instructional tool for CP/M. ◆◆J

(Listings begin on page 114)

CP/M Exchange (Text begins on page 110)

Listing One

```
;
;       DISPLAY DISK CONTROLLER STATUS IN BINARY SUBROUTINE
;
;       THIS SUBROUTINE WILL DISPLAY THE 1793 DISK CONTROLLER CHIP
;               STATUS IN BINARY
;
;       INPUTS: A = VALUE TO BE DISPLAYED
;       OUTPUTS: NONE
;       REGISTERS USED: NONE
;

DSKST           EQU     $
                PUSH    PSW             ;SAVE USER REGISTERS
                PUSH    BC              ;*
                PUSH    DE              ;*

                LD      C,CR            ;OUTPUT CR AND LF BEFORE DISPLAY
                CALL    CONOUT          ;*
                LD      C,LF            ;*
                CALL    CONOUT          ;*

                IN      A,(DSTAT)       ;READ DISK STATUS
                LD      D,A             ;SAVE IN REGISTER D
                LD      B,8             ;NUMBER OF BITS TO DISPLAY

DSKSTR          EQU     $
                LD      A,D             ;CURRENT STATUS BYTE TO REGISTER A
                RLA                     ;HIGH ORDER BIT TO CARRY
                LD      D,A             ;SAVE CURRENT STATUS BYTE
                LD      A,'0'           ;ASCII ZERO TO REGISTER A
                ADC     A,0             ;IF CARRY SET ADD 1
                LD      C,A             ;CHARACTER TO C FOR CONOUT
                                        ;ROUTINE
                CALL    CONOUT          ;PUT CHARACTER TO CRT
                DJNZ    DSKSTR          ;LOOP UNTIL COMPLETE

                POP     DE              ;RESTORE USER REGISTERS
                POP     BC              ;*
                POP     PSW             ;*

                RET                     ;BACK TO CALLER
```

End Listing One

```
;
;         TWITTLE SUBROUTINE
;
;         This subroutine will alternately display an asterick and space
;                 at cursor position
;
;         INPUTS: NONE
;         OUTPUTS: NONE
;         REGISTERS USED: NONE
;

                NAME    ('TWITLE')

                .Z80

                ENTRY   TWITLE

;
;         EQUATES FOR THIS ROUTINE
;

COS             EQU      9              ;PRINT CONSOLE STRING FUNCTION
BDOS            EQU      5              ;BDOS JUMP OFF POINT
BKSP            EQU      8              ;BACKSPACE CODE
SPACE           EQU     ' '             ;SPACE CONSTANT
AST             EQU     '*'             ;ASTERICK CONSTANT
DOLLAR          EQU     '$'             ;DOLLAR SIGN CONSTANT

;
;         ENTRY POINT AND MAIN CODING
;

TWITLE          EQU     $
                PUSH    PSW             ;SAVE ALL USER REGISTERS
                PUSH    BC              ;*
                PUSH    HL              ;*

                LD      A,(INDBYT)      ;GET DIRECTION BYTE
                BIT     7,A             ;CHECK DIRECTION BYTE
                JR      Z,TWITB

TWITA           EQU     $
                LD      DE,TWITIN       ;ERASE ASTERICK STRING
                RES     7,A             ;RESET DIRECTION BYTE
                LD      (INDBYT),A      ;SAVE DIRECTION BYTE
                CALL    PMSG            ;DISPLAY MESSAGE ON CONSOLE

TWITEX          EQU     $
                POP     HL              ;RESTORE ALL USER REGISTERS
                POP     BC              ;*
                POP     PSW             ;*

                RET                     ;BACK TO USER
```

(Continued on page 119)

Listing Two

```
TWITB           EQU     $
                LD      DE,TWITOT       ;DISPLAY ASTERICK STRING
                SET     7,A             ;SET DIRECTION BYTE
                LD      (INDBYT),A      ;SAVE DIRECTION BYTE
                CALL    PMSG            ;DISPLAY MESSAGE ON CONSOLE
                JR      TWITEX          ;EXIT

PMSG            EQU     $
                LD      C,COS           ;PRINT STRING FUNCTION CODE
                CALL    BDOS            ;ALLOW BDOS TO DISPLAY ENTIRE
                                        ;STRING
                RET                     ;BACK TO USER

;
;       CONSTANTS AND SAVE AREAS
;

TWITOT:         DB      AST,BKSP,DOLLAR
TWITIN:         DB      SPACE,BKSP,DOLLAR
INDBYT:         DB      Ø

                END
```

End Listing Two

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

Language Benchmarks

An interesting program to test the speed and accuracy of various high-level languages has been sent to us by Bill Savage of Microfloat in Houston, Texas (see Listing One, page 122). In the 8-bit versions, the CPU was an 8085 at 5 MHz. The 16-bit CPU was an 8086 at 5 MHz. The floating-point hardware versions used the Intel 8232 (8 bit) or the Intel 8087 (16 bit). The final result of each test *should* be 2500.000; any variations from this figure reflect errors in the language's floating-point algorithms or rounding methods. The arctangent in the central function is designed to accentuate such discrepancies and make them more visible. The execution times are listed in increasing order in Table 1 (page 122); the accuracy results speak for themselves.

Although all of the results with the standard software foating-point libraries showed poor accuracy, the PL/I results were especially surprising. This compiler has always had a reputation for high quality and performance. Not only was the result farther off than with any other language, but the code produced by the PL/I-86 compiler was actually slower than the BASIC-86 interpreter! Another interesting observation was the small difference in execution times on this test between compiled and interpreted BASIC, demonstrating that on number-crunching programs relatively little time is being spent in the interpreter.

Optical Mouse Interface

Rick Wilton contributed two short programs which demonstrate use of a "mouse" on the IBM PC. He used the MSC M-1 Optical Mouse sold by Mouse Systems Corp., 2336H Walsh Ave., Santa Clara, CA 95051. The software assumes that the mouse is on the first serial port – COMO.

A BASIC version of the interface is supplied as Listing Two (page 123), and a Forth version as Listing Three (page 126). The BASIC version may be used in the ordinary text display mode, while the Forth program uses the medium-resolution graphics mode. In the latter version, pushing the left button causes the mouse to leave a trail and pushing all three buttons at once ends the demonstration.

The mouse sends out a packet of five data bytes whenever there is a change of mouse state. This change may either be a movement or a switch transition. The first byte of a packet always has the bit pattern 10000xxx, so the driver program can synchronize with the mouse by simply reading and discarding bytes until one with such a pattern is found. The packets then contain the following data:

Byte	Contents
0	1 0 0 0 0 L M R*
1	ΔX_1
2	ΔY_1
3	ΔX_2
4	ΔY_2

*Left, middle, and right buttons.

The mouse supports two types of protocols: Rotatable, and Non-Rotatable. In the Rotatable mode, the position of two sensors is transmitted, which makes it possible for the interface program to determine both the rotational and translational orientation of the mouse. However, in most applications only the translational information is of interest, so the Non-Rotatable protocol is used. The mouse's two sensors then send out redundant information. This also allows continued operation in case of failure of one of the LED sensors.

XSUB for CP/M-86

Mr. Terje Bolstad of Elektrokonsult has sent us a pre-release copy of his company's XSUB utility for CP/M-86. XSUB extends the SUBMIT utility to include input to transient application programs from batch files, analogous to the XSUB which is supplied by Digital Research for CP/M-80. This can be very useful for automatic execution of tasks which require long sequences of commands. For example, you could set up a batch file that would automatically format a new diskette, copy the system tracks, and then transfer selected files without any operator intervention. Digital Research's release of CP/M-86 without a XSUB was the subject of many heartfelt complaints by users who had gotten used to that capability under CP/M-80, and I am glad to see an independent software house filling the gap.

Incidentally, Mr. Bolstad pointed out some "features" of the ordinary SUBMIT.CMD utility that are not mentioned in the CP/M-86 documentation. First, all lines in a submit file starting with a semicolon are treated by SUBMIT as comments; i.e., they are displayed as they are encountered during the execution of a submit file, but no other action is taken. Second, SUBMIT will convert the two letter combination of an "^" followed by an ASCII character A-Z into the corresponding control character. Third, SUBMIT will terminate without an error message if it encounters an empty line in a batch file, even if the blank line is followed by valid command lines.

The XSUB.CMD utility is available from Elektrokonsult A/S, Inc., Konnerudgaten 3, N-3000 Drammen, Norway for $39.00 plus shipping. At present only standard 8-inch SSSD and 5¼-inch DEC Rainbow disk formats are available.

IBM and Epson Printer Control Codes

Dan Daetwyler, who contributed the IBM PC graphics printer dump article in June *DDJ*, has written with some additional information for those who may have run into unexpected problems with it.

He says, "I've learned the hard way that there is quite a variety of ROMs available for the Epson MX80 (and equivalent IBM) printers. The current 'crunch' is primarily the difference between the Epson MX80 with GRAFTRAX option and the 'IBM Graphics Printer.' The following list of differences between the two ROMs may be useful to your readers. I can't promise that only two versions of the ROMs are out in the world, but at least this is a good first step.

"The following DECIMAL codes are only active on the GRAFTRAX ROM and are not available on the IBM Graphics Printer:

	35	– Accept 8th bit as is
	8	– Backspace 1
	136	– Backspace 1
Esc	61	– Clear 8th bit
	127	– Delete last character in buffer
	255	– Delete last character in buffer
Esc	53	– Italics OFF
Esc	52	– Italics ON

"The following DECIMAL codes are only active on the IBM Graphics Printer, and are not available on the GRAFTRAX ROM:

	24	– Cancel the line, clear printer buffer
	152	– Cancel the line, clear printer buffer
Esc	55	– Character set 1
Esc	54	– Character set 2

"The following DECIMAL codes are used on both ROMs but function differ-

ently between the two implementations:

GRAFTRAX:

Esc 65 n – Set line spacing to n/72 inches (immediate)
Esc 50 – Reset line spacing to default (1/6 inch)

IBM Graphics Printer:

Esc 65 n – Prepare line spacing to be n/72 inches
Esc 50 – Set line spacing as prepared by Esc 65 n (If no Esc 65 pending, defaults to 1/6 inch)

"These differences are really minor, but as my article in the June issue proves, can lead to trouble. I set the SHDGRPH program up on a GRAFTRAX printer and was a bit less than pleased when I discovered it wouldn't work properly on the IBM Graphics Printer. The difference is in the Esc 65 sequence, which I used. I preset the line spacing (when running on the IBM), but never implement until the dot matrix printing is done. This has the dual effect of leaving white space between each row of eight dots, and of leaving the printer in an 8/72 inch line spacing at end of job (Ugh!); while if I had written the program for the IBM Graphics Printer, running it on a GRAFTRAX would have printed the graphics at 1/6 inch line spacing (white space again) but at least would have left the printer in a reasonable line spacing mode. Oh well.

"The sources of this data are the Epson GRAFTRAX manual and the IBM Technical Reference Manual (XT version). Both are copyrighted publications, and GRAFTRAX and IBM Graphics Printer are trademarks of the respective vendors."

With hardware floating point libraries from MicroFloat:

Language	Version	Result	Time (sec.)
PL/I-86 with 8087	1.01	2477.244	3.7
8080 Assembler with 8232	RMAC	2499.955	10.2
PL/I-80 with 8232	1.40	2499.995	10.4
BASIC-80 with 8232	5.20	2500.000	10.7
Fortran-80 with 8232	3.40	2499.995	12.5

With standard floating point libraries:

Language	Version	Result	Time (sec.)
BASIC-86 Interpreter	5.20	2179.850	92.2
Fortran-80	3.40	2304.863	140.8
BASIC-80 Compiler	5.20	2304.860	140.8
BASIC-80 Interpreter	5.20	2304.860	174.9
PL/I-86	1.01	1641.758	179.6
PL/I-80	1.30	1641.758	254.4

Table 1.

Comparisons of execution speed and accuracy for commonly available high-level languages, with and without hardware floating point support. The 8087 and 8232 support libraries are marketed under the names FLOAT87 and FLOAT80, which are trademarks of MicroFloat.

Concurrent CP/M-86 Enhancements

Like many other early purchasers of Concurrent CP/M for the IBM PC, I was impressed with the potential power of this operating system but dismayed at its "black box" nature. It seemed that in this case Digital Research had departed from its old policies of providing adequate technical information for third party hardware vendors and systems programmers. A particularly sore point was the way Concurrent CP/M takes over all of the interrupt vectors, thereby precluding any access to the PC's ROM Bios. Another frequently heard complaint was the fact that you couldn't boot up the operating system without a disk in both drives.

Digital Research has now issued three very welcome Application Notes, which are available to registered owners of Concurrent CP/M-86 from the Technical Support Group at P.O. Box 579, Pacific Grove, CA 93950. The first Note deals with access to the ROM Bios, and includes a sample program to make a call to the ROM's Print Screen utility. The second Note provides a program to adjust the horizontal SYNC position of the video controller. The third Note gives the code for a patch to the operating system, installed with DDT86, that allows you to boot CCPM from a single drive.

Digital Research is also making available (to qualified requestors) a set of application notes and example programs for what they term "Field Installable Device Drivers." This allows the programmer to link a software driver for practically any type of peripheral device to the IBM PC's CP/M-86 or Concurrent CP/M-86 operating system with a minimum of fuss, in much the same manner as the user-installed driver capability now available under MS-DOS 2.0. The literature they sent me included sample drivers for a RAMDISK and a Zobex Hard Disk.

Incidentally, the much bally-hooed Digital Research $60 CP/M-86 for the IBM PC has finally arrived. This release was supposed to be The Answer to Digital Research's previous problems in the PC marketplace, and lure back a sizable portion of the users from Microsoft's PC-DOS. Well, the new version of CP/M-86 does include print spooling, a set of GSX graphics device drivers, and some fancy new screen-oriented utility programs for device configuration and disk copying/formatting. The documentation is excellent, though it is a sign of the times that the technical information which used to occupy 90% of the DRI manuals is now relegated to a few appendices.

But when all is said and done, there are really no substantive changes in this new release of CP/M-86, and the performance of the operating system is still just plain slow. Perhaps if Digital Research had included some of the features they added to CP/M-80 version 3, such as time and date stamping of files and hashed directory searches, and thrown in a resident COPY function, they might have come up with a viable competitor to Microsoft's operating system – but I can't believe that anyone who is used to the responsiveness of PC-DOS could ever be satisfied with this incarnation of CP/M-86. I suspect DRI is beginning to realize that, too, for they recently announced PC-DOS versions of all their language compilers.

DDJ

Circle no. 20 on reader service card.

16-Bit Toolbox (Text begins on page 120)

Listing One BASIC and PL-1 versions of Bill Savage's accuracy and speed test for high level languages.

```
100   ' Time and General Accuracy Test Program
110   DEFINT I
120   ILOOP=2500
130   A=1
140   FOR I=1 TO ILOOP-1
150     A = TAN(ATN(EXP(LOG(SQR(A*A))))) + 1
160   NEXT I
170   PRINT USING "A=####.####";A
180   STOP
```

```
actest:

          proc options (main);

          declare   (i,iloop) fixed bin(15),
                    a         float bin(24);

top:      a = 1 ;

          put skip(2) list ('Ending loop value : ');
          get list (iloop);

          if (iloop < 2 ¦ iloop >3000) then goto top;

          do i=1 to iloop-1;

             a = tan(atan(exp(log(sqrt(a*a))))) + 1.0 ;
             end;

          put skip edit ('I=',i,'A=',a) (a,f(5),a,f(14,8));

          goto top;

end actest;
```

End Listing One

Listing Two

```
10 REM  Demonstration program for M-1 Optical Mouse.
11 REM  Assumes "Mouse" interfaced through RS232 port 0.
12 REM  Run program, then calibrate mouse by sweeping it
13 REM  in wide circles until the "star" on the screen moves.
14 REM  To leave a trail push any mouse button, to terminate
15 REM  the demonstration push all three buttons at once.
20 REM  Richard Wilton --- January 1983
30 REM
40 GOTO 1000
50 REM
100 REM initialize RS-232 port for 1200 baud, no parity, 8-bit data
110 ON ERROR GOTO 150
120 OPEN "COM0:1200,N,8" AS #1
130 ON ERROR GOTO 0
140 RETURN
```

(Continued on next page)

```
150 REM come here if comm port initialization times out
155 PRINT:PRINT "comm port initialization failure": PRINT
160 RESUME NEXT
170 REM
200 REM  get a data packet from the mouse interface.
210 REM  synchronize by waiting for a byte with bit pattern 10000xxx
220 IF(INP(P1%)AND 1) THEN B%=INP(P2%):IF (B% AND &H80) THEN ELSE 220
      ELSE 220
225 REM  now read the two sets of x,y coordinates.
230 IF(INP(P1%)AND 1) THEN X1%=INP(P2%) ELSE 230
240 IF(INP(P1%)AND 1) THEN Y1%=INP(P2%) ELSE 240
250 IF(INP(P1%)AND 1) THEN X2%=INP(P2%) ELSE 250
260 IF(INP(P1%)AND 1) THEN Y2%=INP(P2%) ELSE 260
270 IF X1%>=128 THEN X1%=X1% OR C% : IF X1%< -120
      THEN 200
280 IF Y1%>=128 THEN Y1%=Y1% OR C% : IF Y1%< -120
      THEN 200
290 IF X2%>=128 THEN X2%=X2% OR C% : IF X2%< -120
      THEN 200
300 IF Y2%>=128 THEN Y2%=Y2% OR C% : IF Y2%< -120
      THEN 200
310 B%=B% AND 7
320 RETURN
330 REM
500 REM initialize the screen
510 CLS
520 X% = 40 : Y% = 12
530 LOCATE Y%,X% : PRINT "*" ;
540 RETURN
550 REM
600 REM update the screen from XO%,YO%
610 IF B%=7 THEN LOCATE Y%,X%: PRINT " "
620 IF XO% > 0 AND XO% < 81 THEN X%=XO%
630 IF YO% > 0 AND YO% < 24 THEN Y%=YO%
640 LOCATE Y%,X% : PRINT "*"
650 RETURN
660 REM
900 REM set up constants and port addresses
910 C%=&HFF00
920 P1%=&H3FD       ' status port
930 P2%=&H3F8       ' data port
940 RETURN
950 REM
1000 REM demonstration program control starts here
1010 GOSUB 900    ' initialize constants and port
      addresses
1020 GOSUB 100    ' initialize serial port for baud
      rate etc.
1030 GOSUB 500    ' initialize display screen
1040 REM top of mouse tracking loop
1045 REM read next change of mouse state
1050 GOSUB 200
1055 REM update the mouse coordinates
1060 XO% = X% + X1% : YO% = Y% - Y1% : GOSUB 600
1070 XO% = X% + X2% : YO% = Y% - Y2% : GOSUB 600
1075 REM keep going until all three buttons are
      pressed
1080 IF B%>0 THEN 1050
1090 END
```

End Listing Two

16-Bit Toolbox (Listing continued, text begins on page 120)

Listing Three

```
Screen # 80
  0 ( initialize mouse                                          01/29/83 )
  1
  2 HEX
  3
  4 CODE INT14  ( AX DX -- AX )              ( see Tech Ref p. A-20 )
  5  DX POP AX POP  14 INT  AX PUSH  NEXT
  6
  7
  8 : INIT-MOUSE ( -- AX )      ( set commo port 0 to 1200 baud .. )
  9    083 0 INT14 ;          ( .. no parity, 1 stopbit, 8-bit data )
 10
 11 DECIMAL
 12
 13  -->
 14
 15

Screen # 81
  0 ( read data from mouse                                      01/29/83 )
  1
  2 FORTH DEFINITIONS HEX
  3
  4 CODE @MOUSE ( -- dX dY dX dY LMR )
  5       CX, # 3 MOV                                ( # bits for shift )
  6 1$:   DX, # 3FD MOV     AL, DX IN        ( read RS-232 LSR port )
  7       AL, # 01 AND                        ( wait for Data Ready )
  8       1$ JZ
  9       DX, # 3F8 MOV     AL, DX IN       ( read RS-232 data port )
 10       AH, AL MOV                       ( copy to hi-order byte )
 11       AX, CL SHR                  ( put flag in AH, LMR in AL )
 12       AH, # 10 CMP                 ( 1st byte of data packet? )
 13       1$ JNE                              ( try again if not )
 14       BL, AL MOV                        ( save LMR for later )
 15  -->

Screen # 82
  0 ( read data from mouse, cont.                               01/29/83 )
  1
  2       CX, # 4 MOV                                ( count for loop )
  3 2$:   DX, # 3FD MOV     AL, DX IN        ( read RS-232 LSR port )
  4       AL, # 01 AND                        ( wait for Data Ready )
  5       2$ JZ
  6       DX, # 3F8 MOV     AL, DX IN       ( read RS-232 data port )
  7       CBW                                          ( sign extend )
  8       AX PUSH                                ( push onto stack )
  9       2$ LOOP
 10       BH, BH XOR                          ( zero hi-order byte )
 11       CL, # 5 MOV                        ( set # bits to shift )
 12       BL, CL SHR                  ( shift LMR flags to right )
 13       BX PUSH                                ( push the result )
 14       NEXT                                        ( c'est tout )
 15  DECIMAL -->
```

(Continued on next page)

```
Screen # 83
  0 ( demo for optical mouse                                    01/29/83 )
  1
  2 0 VARIABLE LEFT                                  ( button status flag )
  3
  4 : DEMO ( -- )
  5    INIT-MOUSE DROP                  ( set comm port, ignore status )
  6    6 MODE 320 100 2DUP 1 !dot        ( start at center of screen )
  7    BEGIN @MOUSE -DUP WHILE
  8     4 AND 4 = LEFT !                                ( update button )
  9     LEFT @ IF 6 PICK 6 PICK 0 !dot ENDIF            ( erase old x,y )
 10     >R >R  MINUS ROT + >R + R>                           ( 1st x,y )
 11     LEFT @ 0= IF 2DUP 1 !dot  ENDIF                 ( mark 1st x,y )
 12     R> R>  MINUS ROT + >R + R> 2DUP 1 !dot               ( 2nd x,y )
 13                         REPEAT    ( until all 3 buttons are down )
 14    2 MODE 2DROP 2DROP 2DROP  ;
 15
```

End Listing Three

OF INTEREST

by Michael Wiesenberg

Send Your Resume To the "Job Registry"

For years, doctors and lawyers have had the luxury of using their own exclusive "job referral services" to advertise their professional experience to prospective employers. But now, thanks to a small company in Hayward, California, there will soon be available an on-line computer database called **The Software Pro's Job Registry**.

This service has actually been in existence since July 1982; but as the database grew, its founders decided to design the software and hardware necessary to put the entire service on-line. Scheduled to be implemented in February 1984, this service will continue to help programmers match up their skills and interests with *real jobs*. The Software Pro's Job Registry was started by a high-tech marketing and advertising agency called C.J. Street & Associates, and was originally provided only to programmers who understood the Forth computing language. Today, however, this service is extended to all programmers with experience in *any* computing language.

The people who operate this job referral service are now advertising for new members; and, as part of the service, they also publish an occasional newsletter. However, the membership fee is a ghastly one dollar. So, if the price doesn't scare you away, and you're still interested, send a self-addressed, stamped envelope to: The Software Pro's Job Registry, c/o C.J. Street & Associates, 24301 Southland Drive, No. 216, Hayward, CA 94545; or call (415) 887-2687.

Tiny TV and Tinier Computer

Casio, a company that makes all kinds of tiny electronic miracles, has introduced **Casiovision**, the world's smallest (3.19-by-4.69-by-1-inches in dimension, and weighing a mere 12.35 ounces) pocket television. It has a 2.75-inch screen and a display element that is a twisted nematic type of liquid crystal. The screen can be seen equally well in the dark (because of an electroluminescent backlight) as out in the bright sunlight (because of its translucent high-resolution LCD with high duty dual matrix drive system). Casiovision has four modes of power supply: AC adapter, DC (three AA batteries), car battery, and rechargeable nickel cadmium battery. It has an earphone jack and its own speaker, and costs $299. Not only can you hold an ice cream cone in one hand and this tiny TV in the other; but (and here's why I mention it) suddenly the technology for very tiny computer monitors exists.

Casio has also entered the medium-priced notebook computer and hand-held computer fields. The **FP-20** has a 20-column by 8-line display for data and graphic information. It comes standard with 8K RAM (expandable to 32K) and 32K ROM (expandable to 40K), BASIC, a built-in spreadsheet program, a full-size keyboard, RS-232 and Centronics-compatible parallel ports, weighs under four pounds, measures 12.4-by-8.8-by-2.2-inches. The FP-20 runs on batteries or AC, and costs $499. The **FX-802PAC** hand-held computer has cassette interface and a *built in* thermal printer, 2K RAM, BASIC, and costs $189.90. On top of all this, Casio has a software library for the HHC with over 50 programs in prices that range from $15 to $35. **Reader Service No. 103.**

Developing for CROMIX

Model CXDR from **Cromemco** is a development system for systems integrators and others to develop custom I/O drivers for CROMIX – the multi-user, multi-tasking UNIX-like operating system. The accompanying documentation describes how to add character device drivers and explains how primary driver subroutines are handled. It also details utilities to add and remove interrupt vectors, transfer and buffer characters, sleep, wake up, and interface with the TUART. The software can be used with Z80 CROMIX and the 68000/Z80 CROMIX-D dual processor operating system. Cromemco particularly recommends the system for those using the real-time control capabilities of CROMIX, and for those who wish to interface to specialized sensing or electronically controlled devices. $595. **Reader Service No. 105.**

Color It Three Inches

Amdisk III, from **Amdek**, is a three-inch dual disk drive for the Radio Shack Color Computer. It's completely compatible with TRSDOS. The drives provide up to 624K double-density formatted storage for $599; and, they say, "the media is $6.99 each." Or did they mean, "the *medium* is $6.99 each"? **Reader Service No. 107.**

Old Rumor Confirmed

A long time ago, I hinted at the possibility of using CP/M on the Atari 400 or 800. A reader wrote in asking for more information, which I was unable to supply because I had been sworn not to reveal the source of my leak. However, now comes an announcement from **USS Enterprises** (clever, huh, Trekkies?) of the improved **Critical Connection**, an interface that allows the Atari to use the disk drives, printer, and keyboard of any CP/M system with a 19.2 Kbaud serial port. The system is configured for Kaypro, North Star, Sanyo, some Cromemcos, CCS, Heath-Zenith, and Morrow. You get hardware to connect the CP/M system to the Atari, software on various formats, simulation of four Atari disk drives, software buffers for CP/M keyboard and printer, error detection and recovery, documentation, and technical support. You can write and edit Atari programs on either system, and store them on the CP/M disk drives, or print them out on the CP/M printer. $175, and you can get a free brochure. **Reader Service No. 109.**

Public Domain PC Software

Now, you can buy a book that'll show you how to get public domain software for your PC-DOS. The **New**

York Amateur Computer Club (NYACC) has a library out that includes languages, applications, utilities, games, and more, all listed in their **Catalog of Public Domain Software for PC-DOS, Book 1**, one of several catalogs available by mail from the NYACC. This catalog costs $10 in North America and $15 for overseas airmail (U.S. funds). The catalog lists the first 26 volumes in the PC/Blue Public Domain Library, with tables of contents and abstracts of files, and all document files contained on the diskette volumes. To get the diskettes, send $6 (per diskette) to: NYACC, P.O. Box 106, Church St. Station, New York, NY 10008. Also available at the same price is the **Catalog of Public Domain Software for CP/M, Books 1 through 8**. Each lists the contents of scores of volumes, with such goodies as software tools from Australia, programs from the PL/I and Pascal Z Users' Groups, 68000 cross assembler, an update of MODEM7, as well as CP/M-86 utilities, a Z80 Command Processor Replacement, and more . . . including contributions from SIG/M. **Reader Service No. 111.**

Standard Bulletin Board

The Commerce Department's **National Bureau of Standards** has established a computer bulletin board, the **Microcomputer Electronic Information Exchange (MEIE)**, for topics about micros. This bulletin board provides information on conferences, seminars, workshops, telecomputing services, user and special interest groups, other CBBs, publications, new computer products, and technology. You don't have to be part of the government to use the bulletin board, and it's free; all you need is an ASCII terminal, with eight data bits, no parity and one stop bit. Call (301) 948-5718 *any* time to access the service, or (301) 948-5717 to reach a bulletin board devoted to exchanging information on computer performance evaluation topics. **Reader Service No. 113.**

Lots of Shows

Northeast Expositions announces the following computer shows:

The **Second Annual Twin Cities Computer Show and Software Exposition** will be held September 15-18 at the Minneapolis Auditorium in Minneapolis, Minnesota. This is "the largest annual computer exposition in the north central states, and features 350 displays." Daily admission is $5 for adults, $3 for children. **Reader Service No. 115.**

The **Second Annual Rocky Mountain Computer Show and Software Exposition** will be held September 22-24 at the Denver Merchandise Mart. This is "the largest annual computer exposition in the Rocky Mountain states, and features 350 displays." Daily admission is also $5 for adults and $3 for children. **Reader Service No. 117.**

CP/M '83 East will be held September 29 - October 1 at Boston's Hynes Auditorium. The Exposition portion of the event has been said to be "the largest presentation of CP/M hardware and software ever assembled." The Conference Program features nearly 100 sessions, including such "noted leaders from the software industry" as Gary Kildall, Sol Libes, Stewart Alsop (he's in the software industry?), and Benjamin Rosen. Tickets are $25 for three days of exhibits and conferences, or $10 for a one-day ticket for the exhibits only. **Reader Service No. 119.**

And, while you're in the area, **PC '83** will be held October 8-10 at Boston's Bayside Exposition Center. Tickets are $25 for three days, or $10 for a one-day, exhibits-only ticket. **Reader Service No. 121.**

Meanwhile, on the other side of the country, **Applefest/San Francisco**, the largest Apple computer show in the U.S., will be held October 28-30 at the Moscone Center. Events will include an open forum with Steve Wozniak and a panel discussion with software experts on "The Great Software Piracy Debate." Tickets are $25 for three days of exhibits and conferences, or $10 for one day of exhibits only. **Reader Service No. 123.**

And, if you're still hanging around Boston, the **Fifth Annual Northeast Computer Show and Software Exposition** will be held November 10-12 at Boston's Hynes Auditorium. This, they claim, will be the "largest annual end-user computer event in the East." (This falls into the same category as such statistics as "most home runs hit in the third inning of the third game of the World Series by a switch-hitter against a left-hander with the sun in his eyes and the score tied.") There will be approximately 500 displays and exhibits. Daily admission is $7.50. **Reader Service No. 125.**

The **Fourth Annual San Diego Computer Fair**, sponsored by the San Diego Computer Society, will be held November 5 and 6 at the Scottish Rite Center. Registration is $5; there's a banquet, $12; or both for $17. The banquet will feature a speech by Wright State University engineering department researcher Dr. Jerrold Petrofsky about his computerized method of transmitting electrical impulses to the muscles of paraplegics. **Reader Service No. 127.**

Software Maintenance Workshop

A **Software Maintenance Workshop** will be held from December 6 to 8 at the Naval Postgraduate School in Monterey, California. The workshop is being sponsored by the Technical Committee on Software Engineering, the National Bureau of Standards, and the IEEE Computer Society. **Reader Service No. 129.**

XFORTH XCHANGE

A small company in Little Rock, Arkansas called "**HAWG WILD Software**" recently announced a new "forum for the exchange of ideas" among users of their new XFORTH, Forth-79 software package. "No idea is too small, too large, or too unusual," claims a yellow index-card-sized press release that we received in the mail last week. The service is free and unrestricted. Send your questions and implementations to: XFORTH XCHANGE, c/o HAWG WILD Software, P.O. Box 7668, Little Rock, AR 72217.

(Continued from page 10)

other off-the-shelf applications. Many of these programs depend on efficient data retrieval rather than sequential file transformations.

If CP/M Plus had taken into account the special cases that arise during program development, would the OS run significantly faster than it does? I doubt it. Some general rules for optimizing the performance of operating systems are: concentrate on improving those things that are slow, not those that are already efficient; solve the general problem, not specific instances; and optimize the execution of individual, time-consuming programs, instead of trying to anticipate future command sequences. Here's what CP/M Plus does to improve the performance of application programs:

First, CP/M 2.2 disk delays fall into three categories. The worst bottleneck comes when searching disk directories. Relatively little data are transferred, but moving the head and scanning multiple sectors to find the file control block make this operation very slow. CP/M Plus implements a fascinating scheme of hashed directory tables and a separate pool of directory sector buffers to eliminate most of this activity.

A second, lesser bottleneck results when programs do a lot of direct record accessing, which requires head movement and rotational latency delays. This is where the CP/M Plus LRU buffering scheme comes into play, by eliminating repeated accesses to particular sectors. Statistically speaking, you *can* generally enhance the performance of direct sector accessing by allocating more buffer control blocks. This does not mean each additional buffer decreases disk activity by some measurable amount.

The third category, for which CP/M has always been relatively fast, is reading and writing logically sequential records. The CP/M record pre-allocation scheme already minimizes head movement and rotational latency in this case. LRU buffering cannot, in general, improve performance for sequential files, since reading or writing a particular record virtually guarantees that a lot more disk activity will occur before the same record is accessed again. Of course, the record may never be accessed again, and any time consumed when buffering it would be wasted. Memory caches are normally small relative to the potential combined length of the files generated during a work session. If so, it makes perfectly good sense to me to reserve sector buffers for when they will do the most good, instead of wiping them clean for a single file.

One way sequential file performance can be improved, however, is by prefetching successive records when a disk sector is read. In essence, CP/M Plus does exactly this when multiple logical records are blocked into each physical sector. CP/M Plus can greatly improve the reading or writing of multiple sequential records, but for this the host system hardware must be able to transfer a series of sequential or adjacent physical sectors, and the application program must be designed to take advantage of the new capability.

I can understand and sympathize with Mr. Cortesi's disappointment that adding a lot of memory cards didn't seem to help his system's performance. But which of the CP/M Plus performance improvements was he testing? Not directory searches, because his tests opened just a handful of files. Not direct sector accessing, because he admitted, "We haven't tried any direct-access I/O yet."

The test programs that he described all loaded sequentially, read input files sequentially, and wrote output files sequentially. Mr. Cortesi found that these programs ran essentially as fast as they had under CP/M 2.2. The column doesn't describe the CP/M 2.2 system used for the comparison, but I would guess the sequential optimization of record blocking and deblocking is implemented within its BIOS. If so, I am not surprised that both systems perform the same.

The engineers at Digital Research could have chosen not to override the LRU scheme for sequential file accesses. This probably would have improved the operating system's apparent performance somewhat when assembling short (sequential) source files. I'm not too concerned, though, with optimizing the performance of programs that only take 25 seconds to complete anyway. It's much more intriguing to me that CP/M Plus can reduce an eight minute PL/I compilation to five minutes, or sort and merge a dozen raw data files with an existing database program in four minutes instead of fourteen.

Near the end of the column, Mr. Cortesi makes some suggestions for further improvement. One is to encourage the retention of COM files in memory. CP/M Plus can buffer the entire CCP.COM file (which executes about as often as all others put together) in a reserved portion of RAM, so it reloads instantly. The OS loads other COM files using the high-speed multiple-sector I/O capability.

Mr. Cortesi also outlines an alternate LRU scheme that buffers all sectors written sequentially, but would be overridden for sequential reads. "Eventually the input requests would work around to the sectors that had not been displaced," and from there on, "operate at RAM speeds," he claims.

Not quite. Output produced while reading in the early sectors would overwrite the remaining sectors, producing the same undesirable ripple effect that discards each sector just before it's needed. I strongly suspect that neither this scheme, nor the proposal of heuristically shifting sector buffers between input and output pools, would make much difference in the long run for most real-world applications.

Despite these objections, I fully agree with Mr. Cortesi's final conclusion, which seems to be that CP/M Plus is worth having, but one should not go overboard in allocating data sector buffers.

CP/M Plus has many advantages over CP/M 2.2 aside from performance. Perhaps in future columns Mr. Cortesi could give us his impressions of the more versatile CCP, automatic diskette log-in, alternate drive search chains, file time and date stamping, implicit execution of submit files, a larger TPA, the improved documentation, the on-line HELP facility, and so on. I'm sure your readers would find his views on these topics most interesting.

Sincerely,
John H. Wharton
Applications Research
P.O. Box 2038
Sunnyvale, CA 94087

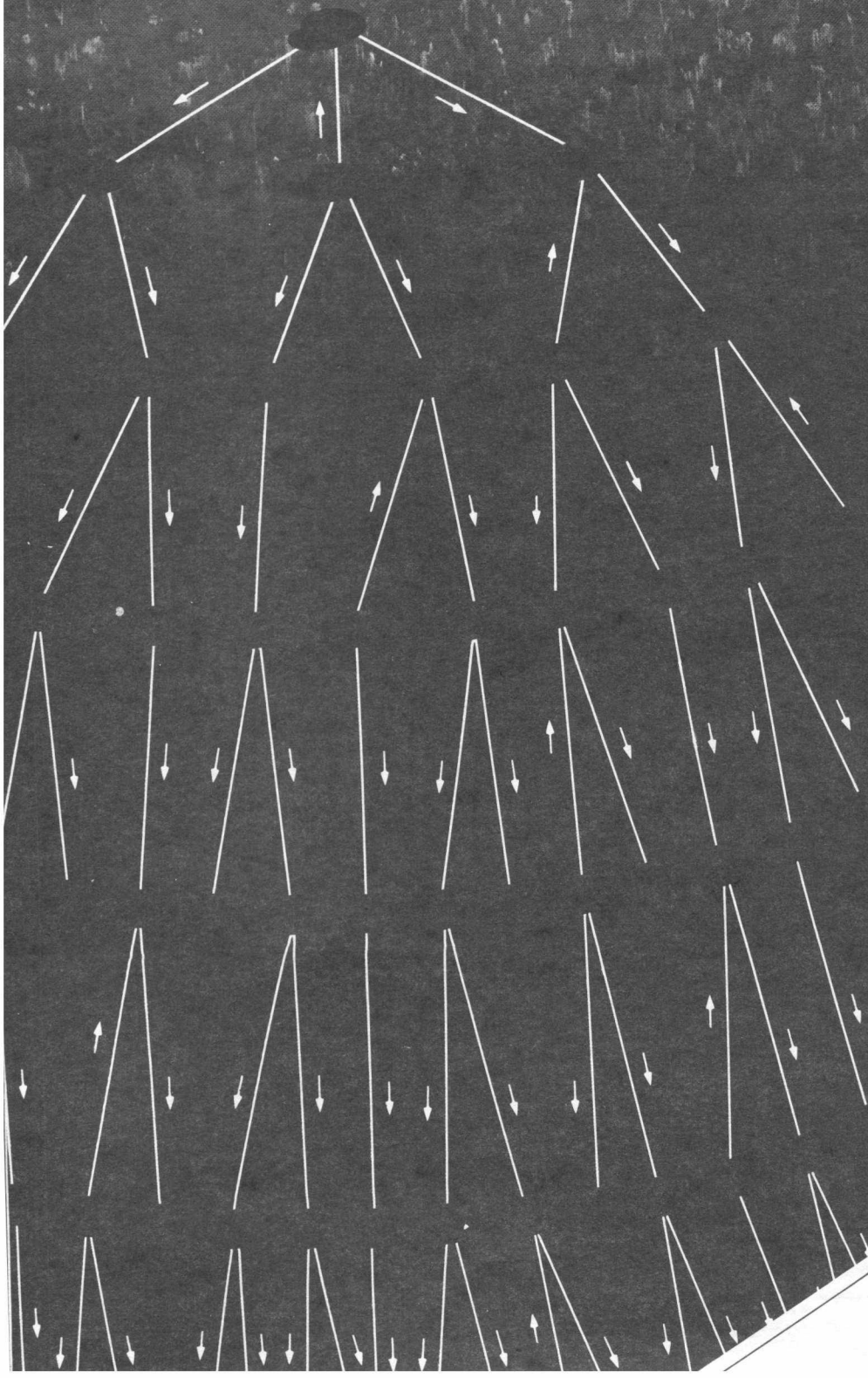
DDJ

#84 October 1983

$2.95 ($3.50 in Canada)

Dr. Dobb's Journal

Ten Years of Small Computer Software



UNIX TO CP/M

Winchester to S-100

Small-C Help Facility

Epson Bit Plot Graphics

NEW! Introducing:
The C/UNIX Programmer's Notebook

Publisher – Jane Nissen Laidley
Editor – Reynold Wiggins
Managing Editor – Craig LaGrow
Contributing Editors –
Robert Blum, Dave Caulkins,
Dave Cortesi, Ray Duncan,
Kim Harris, William Ragsdale,
Michael Wiesenberg
Marketing Director – Beatrice Blatteis
Marketing Assistant – Sally Brenton
Advertising Director – Carl Landau
Advertising Sales –
Alice Hinton, Walter Andrzejewski
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Billie Greenwood, Lew Griffin,
Terri Pond
Production Manager – Barbara Ruzgerian
Production Assistants –
Alida Hinton, Tom Martin,
Shelley Rae
Typesetter – Paula Fairchild
Cover Illustration – Inge Infante

Copyright © 1983 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Christine Bell, W. D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, John W. Campbell, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Thomas Davis, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III, Infoworld, Stan Veit, Western Material Control, S. P. Kennedy, John Hatch, Richard Jorgensen, John Boak. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

Dr. Dobb's Journal
For Users of Small Computer Systems

October 1983
Volume 8, Issue 10

CONTENTS

ARTICLES

20 UNIX to CP/M Floppy Disk File Conversion
by W. E. Wilson
The increasing availability of good, reasonably priced C compilers for use on microprocessor systems has created a large group of C users in the CP/M community. Since C was specifically developed for Unix, however, there is a large amount of C software available for Unix systems. This article describes the structure of Unix files and a program to read Unix floppy disks under CP/M.

40 A Small-C Help Facility
by John Staneff
If you are not very familiar with a given system configuration, on-line help facilities can make operation much easier. Such facilities, in fact, are now included as part of many operating systems. Using a simple ASCII text file to hold the messages, this author shows how to install a hierarchical help system written in Small-C.

71 Attaching a Winchester Hard Disk to the S-100 Bus
by Oscar Goldman
It is now becoming possible to purchase a multi-megabyte hard disk for a less-than-astronomical price. Smart controllers on these devices can greatly simplify their installation in a computer system. The author describes how he linked one such hard disk to an S-100 bus with a small driver and a surprisingly simple interface circuit.

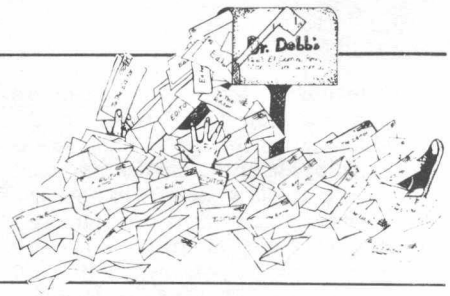
82 Using Epson Bit-Plot Graphics
by Don Gay
Facing the problem of how to generate plots of electron density maps on a dot-matrix printer, this author has designed programs which will calculate the various probabilities and then print them on an MX-80. Also included is a discussion on how to control the Epson and an example of the images which can be generated.

DEPARTMENTS

8 Editorial
9 Letters
12 Dr. Dobb's Clinic
Your fantasy computer system, the buffered keyboard, and being standard.
16 C/Unix Programmer's Notebook
DDJ's new bi-monthly column! Includes a proposed standard for C code layout and incompatibilities in runtime libraries and linkage formats.
98 16-Bit Software Toolbox
Some news, 8086/88 function macros, and more *IBM PC* low-res graphics.
112 CP/M Exchange
"Hidden Treasures," auto disk format selection, and CP/M Plus device tables.
118 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to People's Computer Company, Box E, Menlo Park, CA 94025 · 415/323-3111
ISSN 0278-6508

LETTERS



Grigonis Reconsidered

To the editor:

When I first undertook to challenge certain points made by Richard Grigonis in his article, "Fifth Generation Computers," (*DDJ* #74) with my own article, "Comments on 'Fifth Generation Computers'" (*DDJ* #77), little did I realize what I was getting myself into. No one at *DDJ* warned me that dissenting writers *must* do battle with superhuman intellects! Grigonis's latest article, "And Still More Fifth Generation Computers," (*DDJ* #82) is utterly devastating. Upon seeing an advance copy of the manuscript sent to me by Grigonis himself (no doubt accompanied by a few chuckles) I now find myself much in the situation that the character of Turnus found himself in Virgil's *Aeneid,* when he attempted to do away with Aeneas with one mighty blow, only to find that his sword merely shattered in his hands!

One rarely enjoys recanting one's ideas made in print, and I must grudgingly admit that Grigonis has pretty much blown my arguments to bits, although I still think that he has not adequately addressed the question of program validation in his actor-based language/program generator. No matter – I dare not taunt him or he will undoubtedly wreak havoc once again upon my carefully thought-out arguments.

The first part of the article, which I think amazes me more than anything I have read in *DDJ* (or anywhere else) in a long time, is a succinct, astute analysis that ranges over everything from soup to nuts. First he cleverly makes a fool out of me by pointing out that the speed of light dictates that *all* computers of increasing speed must ultimately be either microcomputers or *the size* of microcomputers, then he goes on to casually mention what the ultimate limits of computation are in the physical world (as if we had known all along), then he literally humiliates the Japanese in a brilliant analysis revealing a "fatal flaw" (namely, the PROLOG language and the predicate logic knowledge representation scheme upon which it is based) in their billion-dollar effort to produce their own version of a fifth generation computer, and *then* he plausibly describes his own ideas for *both* computers and the workstations for the next half century with such clarity and detail that I think he must have a crystal ball powered by a 16 MHz Motorola 68000!

But before we hastily give Mr. Grigonis a MacArthur Award for his sheer intellectual virtuosity, there are a few remaining items that I would still like to challenge him on – although (understandably) I do so with the utmost apprehension!

The PROLOG language may not be as bad as Grigonis thinks. Since last year there have been three major revisions in PROLOG (nobody will say what they are, however) and the Japanese are working on some kind of processor using the molecules in a crystal lattice as flip/flop switches, something roughly along the lines Grigonis suggests when he talks about "optical" microprocessors. An ideal processor based upon a new architecture would execute PROLOG statements *directly* (like the MicroPascal Engine) instead of first compiling all the statements to machine code.

Another reason for using PROLOG, I think, is that whereas LISP programs are very formula-like, and writing artificial intelligence programs in it is more of an art than a science, programmable logic is modular and large numbers of people can work on a giant AI program simultaneously. This fits in well with the Japanese work ethic, where individual creativity is sublimated to that of team work.

I myself am interested in Forth as a language, since it runs much faster than LISP, and is extensible so it could in theory be given LISP-like functions. But logic seems to dictate that LISP is the correct language for use in artificial intelligence. I would be interested in seeing Grigoinis's thoughts upon the matter (*Gulp!*).

It's interesting that Grigonis should pick an arch as the basic shape for his workstation, since the arch is an important symbol or "archetype" (no pun intended) in Jungian dream analysis, wherein it represents a major change or revelation. For this reason artists subconsciously use it in their work. For example, at the climactic end of the motion picture *Mildred Pierce,* Mildred and her husband walk off into a sunrise that is pouring through an arch in the distance. There is, in fact, a whole "dream-like" quality to the exterior design of the Grigonis workstation. Whether Grigonis designed it that way to lull the user or whether he literally dreams up these designs remains to be seen!

Grigonis and I are definitely in total agreement on one thing, and that is the disturbing sociological effects of superintelligent computers and the information explosion. But whereas I have grave fears for the well-being of America, Grigonis seems to take it all in stride. And this leads me to the most puzzling and disturbing aspects of his article found in its concluding paragraphs.

The first part of the article is pretty much upbeat, and Grigonis's arguments are so extraordinarily simple that they border on genius. It's the "2000 AND BEYOND" part at the end that's got me worried. After reading the first part of the article, I expected that the last section was going to be an equally optimistic view of how the world would enter some kind of wonderful technological utopia as a result of the development of the advanced hardware and software that he describes.

Much to my horror, Grigonis's logical progression leads him to make some rather disturbing statements regarding dehumanization or "robotization" (to borrow a term from the late Arthur Koestler) and other similar nasty effects on civilization as we know it. I am in agreement with some of these points and would like to refute some of the others, but I now have some experience in what happens when somebody tries to refute Richard Grigonis!

So, on the one hand, I would like to take my hat off to Grigonis, but on the other, I think that perhaps he would secretly prefer a world scenario on the order of that found in the movie, *Colossus: The Forbin Project!*

Sincerely,
Michael J. Doherty
334 South Maple Avenue
Glen Rock, NJ 07452

Sleight-of-Hand Has Slight Problem

Dear Doctor:

The June '83 *DDJ* finally worked its way around the loan loop to me. (I really ought to get a subscription....) I read with some interest Do-While Jones's article on how to trace through the BDOS despite the problems I pointed out in my Feb. '83 letter on the topic. I must give him/her credit for a good trick, but unfortunately (as a great many of those who tried it have undoubtedly discovered), it will not always work.

The "two systems in memory at once" sleight-of-hand trick is very similar

to the technique I use to debug device drivers, and is subject to the same restrictions. Namely, no "magic" of any sort is allowed to occur in the debugger system. Basic vanilla CP/M systems will not have any trouble, but "high performance" systems that use interrupts, memory management, and possibly DMA will experience anomalous behavior. Consider the problems of having two copies of a BIOS trying to field the same interrupt. Obviously, only one will really get it. The other will probably hang. An even nastier situation occurs if you have memory management hardware that overlays part of the TPA to implement a TURBO or MEMDISK buffer caché. If the second copy of the system happens to be switched out at the wrong time.... A variant of the above problem may also occur with certain portable systems. Because the machines always have a full complement of memory, they do not need to distribute MOVCPM or source to the BIOS. This makes it difficult to build the two copies needed. If you could manage to build the two systems, the second would probably not work because the RAM/ROM banking (used to maximize the size of the TPA) could not be relocated.

The "two-at-once" trick will work quite nicely given that you know enough about the insides of your system to guarantee that there will not be any (adverse) interaction between two copies in memory simultaneously.

Stephen M. Kenton
Software Consultant
475 College
Norman, OK 73069

Fast Divisibility Algorithms Revisited

Editor, *DDJ*

The letter submitted by Prof. Grinstein in *DDJ* #82 has one untestable proposition: that my note on divisibility in *DDJ* #80 would have displeased Archimedes and Newton. Another proposition, that mathematics has many followers, is imprecise since "follower" is not defined. The derogatory tone of the letter suggests that it includes Dr. Grinstein, all mathematicians, and perhaps all who've had a course in number theory, but does not include me! I am not offended to learn that I am part of the much larger set of interested amateurs, which probably includes many other *DDJ* readers. John Kahila wrote me a friendly letter, pointing out that divisibility algorithms are indeed of ancient origin, and were fully generalized by Pascal, Fermat, and Gauss. However, it's likely that no one has yet used these abstract principles to devise working algorithms for divisibility by many (let alone *all*!) prime numbers, since they would be very complex and not very useful. The purpose of my very brief article was to arouse interest in a few easy algorithms, for which someone might find new uses. I am unable to comprehend why it should elicit such an emotional response. The purpose of this reply is to encourage my fellow amateurs to explore possible uses of elementary math without fear of attack by lower echelons of the mathematical elite (its Archimedian levels will simply ignore such trivia!). I also wanted to express my hope that the editors of *DDJ* will not be cowed into subjection of amateur contributions to the *imprimatur* of censors.

Sincerely,

H. T. Gordon
College of Natural Resources
U. C. Berkeley
Berkeley, CA 94720

Expanding Upon Gordon's Fast Divisibility Algorithms

Dear Dr. Dobb's:

I just read H.T. Gordon's article "Fast Divisibility Algorithms" in a friend's copy of the June 1983 issue (#80). The article is well-written and interesting; it also illustrates to some extent the perils of trying to do a literature search in an unfamiliar field.

The divisibility algorithms mentioned by Gordon are not new (as Gordon himself suspects). According to L.E. Dickson (*History of the Theory of Numbers, vol. I, ch. XII;* Chelsea Publications, 1971), the first hint of the divisibility-by-7 test appears in the Babylonian Talmud, where it is noted that 100a+b is divisible by 7 if

(Continued on page 123)

EDITORIAL

This month, Anthony Skjellum presents his first installment of the C/Unix Programmer's Notebook – our new bimonthly column dedicated to the C language and its derivatives, and to the Unix operating system and its look-alikes. The scope is intended to be broad, and, as with all *DDJ* columns, reader interaction and contributions are encouraged. We are pleased to welcome Anthony aboard as a columnist.

* * *

Several months ago this space contained discussion of some issues involving the nature and purpose of public domain software. Recently we were presented with another situation which could fuel quite a few ethical debates. . .

The situation could have happened in any number of computer clubs. A large user group had accumulated a rather sizable library of public domain software. Members were allowed to copy disks at meetings, or they could purchase disks from the librarian. In fact, the sale of these disks produced a substantial portion of the club's income.

One day an ad for public domain disks was noticed in a national periodical. The discription sounded like the club library. Investigation revealed that the advertised disks were essentially the club disks with the references to the club's identity removed. It was further discovered that the person placing the ad was a club member and a member of the club's executive committee. There is no indication that he had appropriated any club materials or equipment. Presumably he had simply made his own set of master copies of the library and was selling duplicates made using his own time, materials, and equipment.

What, if anything, has he done wrong here? The programs were in the public domain, so there is no issue of cheating the authors of the programs. There is no indication that he took anything of material value from the club. His customers would not be club members, so he was not taking any income away from the club. The club had not lost a thing and the software would get wider circulation.

Even so, the library was apparently collected and organized through the unpaid, voluntary efforts of the members, and he was using the fruits of those efforts for his own profit. Did he violate the expectations of those members? It does not appear that he added any value to the software that he sold, and he concealed its origin. Were his customers thereby deceived in any way?

The more we look at the situation, the more sides there seem to be. One central question here involves one's view of the purpose of public domain software. The other members of the executive committee, from their perspective, felt the concept of Public Domain had been betrayed. The member in question, however, felt that he had done nothing wrong. In fact, he felt that he was acting in the spirit of Public Domain by promoting the spread of the disks at the same price that the club was charging.

This series of events offers a chance to examine some fundamental issues about public domain software. Fixing guidelines and boundaries in this area can be tricky at best. The floor is open for debate. Readers, how do you see it?

Reynold Wiggins

DR. DOBB'S CLINIC

by D. E. Cortesi, Resident Intern

Intemperate Editorial Opinion

We received a most formidable envelope in the mail early this summer (see Figure 1, page 13). Brown manila paper, window, cryptic directives to "Postmaster" and "Recipient" – the sort of letter that is either going to be an income tax refund or a draft notice. What was it? Inside was a cheerful offer to let us renew our subscription to *PC* magazine!

Words failed us, or at least printable ones did. There is presumably no law against emulating the appearance of a government mailing, but there darn sure ought to be. As it dawned on us how we had been suckered into opening the envelope, we thought that this must be the most outrageous piece of commercial advertising we'd ever read.

But no, the contents of the envelope actually displayed more sheer brass than its exterior: "Renew your subscription today at current low rates and avoid possible price increases in coming months." Price increases? *Price increases?* For *PC*, the magazine with more advertising space sales than any other in the industry? Listen, they could pay *you* to take that magazine and still make a handsome profit! Price increases my rusty scalpel. . . .

We did not renew our subscription to *PC*.

If I Were a Rich Person. . .

One of our favorite recreations is reading the ads in computer magazines, choosing the components of our ideal computer system. Do you do that? Grand, let's indulge in a group fantasy.

Here's the scenario. Your eccentric uncle, Backus "Bits" Naur, cast off this mortal coil on June 30th. He left you a bequest of $9,999, but he attached the conditions that you must spend at least half of it on a new computer for your personal use, and that you must assemble the system within six months of his death. It's a great opportunity, but you realize that whatever you buy will have to last you for some years. You will want the most quality, capacity, and flexibility you can get. On the other hand, the less you spend, the more change you'll have for other things (like placating your spouse).

There is the challenge: how good a system can you build for a price between $4,998.50 and $9,999.00, based on components that you could obtain by December 31st? Be specific about makes and prices. We will publish the gestalt results early next year.

The Buffered Keyboard, Again

In the May Clinic we told about our problems using an interrupt-driven, buffered keyboard under CP/M. We made one mistake and overlooked one obvious fix. The mistake was in reporting the results of tests under CP/M 3.0 as if they were true of CP/M 2.2; they weren't. Here's the truth of the matter.

The problems arise when you try to type ahead of the system. The typed bytes accumulate in the BIOS's keyboard buffer and are ready whenever the system wants them. However, under CP/M 2.2, the BDOS polls the console frequently to see if a key has been hit. The poll is done prior to each byte of console output, and perhaps at other times. If a key has been pressed, the BDOS reads the byte and examines it. If it is a control-S, output to the screen is halted until another character is typed.

If the byte is not a control-S, the BDOS tucks it away in a one-byte buffer of its own, and ceases to do the polling during output until a console input request has emptied that buffer. This is why control-S won't work unless it is the very first keystroke given. Hit any other key and you have filled the one-byte buffer and caused the poll to be bypassed until the next input request from a program. (Incidentally, a program that calls the BIOS directly for console input will never see that BDOS-buffered byte.)

We *sometimes* lose the leading byte (and only the leading byte) of a typed-ahead line under CP/M 2.2. The reason for this was explained by Allan Bomberger. "Many programs do a one byte read to clear the one byte buffer in the BDOS before they return to location zero. PIP does it. ASM does it. The solution is to begin lines with control-X while typing ahead. The control-X goes into the BDOS buffer and stops the BDOS from polling the console. If the program reads one byte, it reads the control-X. If a program doesn't read the control-X, it is processed by the BDOS as 'erase line' and has no effect as the first character."

Typing a control-X as the first byte of a typed-ahead line works very well under CP/M 2.2. So does typing control-U (suggested by L. Barker) or typing a Return (Nick Hammond). Thanks to all three for pointing out this fix.

One problem remains unfixed and probably unfixable: that some programs take any pending console input as an abort signal. Some are worse than others. Bomberger says, "The CCP is not nice. It polls the console all the time during many of its operations. It does not buffer what it reads. A patch to the CCP will stop it from polling all the time during ERA and DIR and TYPE, etc., but this means that you are unable to stop those commands prematurely."

That's CP/M 2.2. When we claimed that we lost one typed-ahead byte for every byte of console output, we were reporting what happened under CP/M 3.0. Nick Hammond of Torrens, Australia noticed the same thing. "CP/M Plus eats a character for every character it displays. The sequence

pip b:=a:somethin *CR* stat b: *CR*

will eat the 's' of the stat command in user 0, but the 'st' in user 1 because of the extra prompt character. Pipping a list of files clears the buffer since one is swallowed for each character of the filenames displayed."

How To Be Standard

Everybody talks about ASCII, but very few people really understand it. Sure, it's a "standard" for the encoding of characters as binary values. But what is a standard, really? Who says it's a standard, and what does this standard actually say?

Well, it's a standard because the American National Standards Institute sanctions it. And you, too, can find out what the ASCII standard really says for yourself. There are three standard documents that define what ASCII is, and they go far beyond naming the patterns of bits it uses. They are interesting technical documents in their own right, and they should be on the desk of every responsible systems programmer.

The first document is *American National Standard Code For Information Interchange* (ANSI X3.4). This 20-page pamphlet defines the ASCII code values and summarizes their intended uses. Its most interesting parts are its appendices. Appendix A is a careful explanation of the design tradeoffs that were made in defining the code set. Did you ever wonder what the FS, GS, RS, and US bytes were for? Appendix A will tell you. Appendix B, "Notes on Application," is the grabber. Is the CP/M convention of ending each line with CR, LF a standard use of ASCII? Is it more or less standard than the UNIX convention of using LF alone? You can find out by reading Appendix B.

The second document has the unlovely title of *American National Standard Code Extension Techniques for Use with the 7-Bit Coded Character Set* (ANSI X3.41). This document specifies the standard ways in which ASCII (a 7-bit code) can be extended. There are two dimensions: it can be embedded in an 8-bit code; and it can represent an extended set of graphic or control symbols. Or both, since methods are shown for implementing an indefinite number of graphic and control sets using either 7-bit or 8-bit codes. This is the document that makes it clear how nonstandard most personal computer displays are, and reveals how they could all have been compatible while retaining each vendor's unique graphics.

The third document is becoming more interesting every day, as we do more and more teleprocessing. It has another run-on title: *American National Standard Procedures for the Use of the Communication Control Characters In Data Communication Links* (ANSI X3.28). This book amounts to a graduate-level course in telecommunications. It specifies the exact function of the ASCII control characters when transferring data over a phone line. Each of them has a function; do you know what they are? But the best part is the many, many sample protocols that are laid out – complete with procedures for handling timeouts, transmission errors, every possible mishap that can strike a phone line. Protocols are given for both synchronous and asynchronous hardware.

This is not an easy book to read, but if you have to design a teleprocessing system, it will repay your efforts richly. We drew on it to devise a simple protocol for sending files from a CP/M machine to an IBM PC over a 20-foot serial cable; it was a great help. Incidentally, did you realize that the XMODEM protocol is *almost* ANSI-standard? XMODEM opens the communications link by sending a NAK from receiver to sender. Had Ward read X3.28, he'd have probably done it by sending an ENQ from sender to receiver, with a response of ACK to begin transmission or a NAK to say "try me later." That's the function of ENQ.

These books are available from the American National Standards Institute, 1430 Broadway, New York, NY 10018. Their prices are

X3.4	(ASCII code)	$ 5.00
X3.41	(code extensions)	8.00
X3.28	(communications)	16.00

There is a handling charge of $5 on each order totalling less than $50, and you must send a check with the order. It's a worthwhile investment.

Pass The Hat, Again

You may have noticed that the last few Clinics have had a lot of stuff generated by the Intern. That's because you aren't contributing! One of these months the mailbag and the Intern's mind are going to come up empty at the same time, and there won't be a Clinic. We need problems, discoveries, bugs, triumphs, puzzles, pitfalls, gripes – anything. Some of our best topics have come on postcards and in hand-written, one-page letters. What to you was a trivial discovery might save somebody else many hours; what to you is an intractable problem will almost certainly provoke multiple, elegant solutions from the readership. *Write!*

DDJ

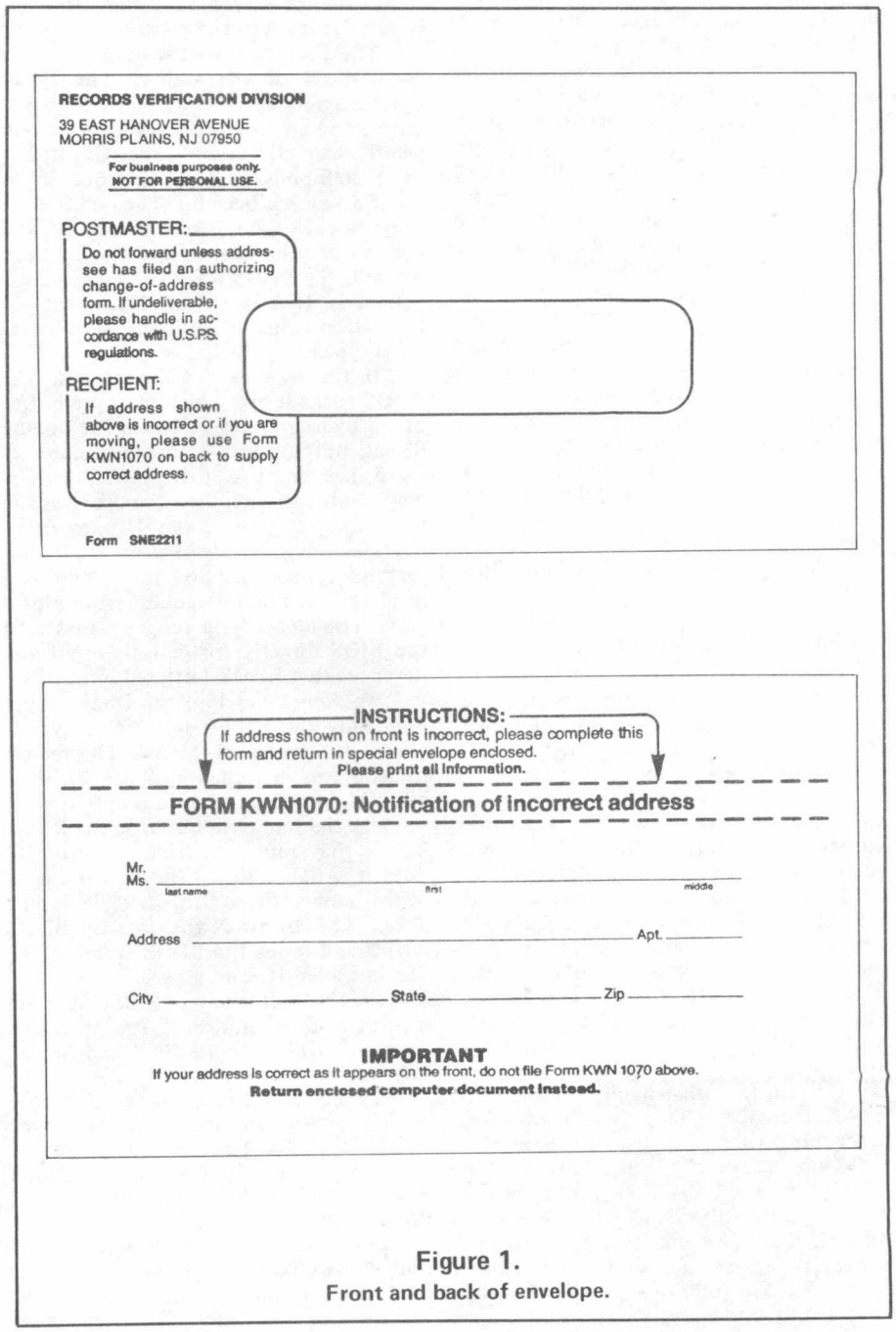


Figure 1.
Front and back of envelope.

C/UNIX PROGRAMMER'S NOTEBOOK

by Anthony Skjellum

This column will deal with the C programming language and the UNIX operating system. These two software systems help programmers produce maintainable and portable code. While C is widely supported on microcomputers and clearly merits discussion, UNIX is only beginning to become available and then only on more expensive microcomputers. Nevertheless, as UNIX design concepts begin filtering down into the more conventional operating systems, discussing UNIX features, successes, and shortcomings becomes both interesting and worthwhile.

This column will also include discussion of software design approaches and current trends. Specific features of C and UNIX will be discussed, as well as more general programming concepts. This will make some of the material presented here pertinent to many types of computer systems and languages.

Readers are invited to participate in this column. I hope that some of the topics will spur readers into producing responses, new ideas, and even new code. Since this column will be published bimonthly, readers will have plenty of time to respond to the topics covered in any given issue.

We start this first column with a discussion of the layout of C code and move on to a discussion of runtime libraries and linkage compatibility for several 8080/Z80/8086 compilers. In the next column, we will discuss how UNIX-type environments tend to produce noninteractive programs. I look forward to your responses and suggestions for future topics.

Layout of C Code

The C language is one of the few standards that apply to a wide variety of computers. *The C Programming Language*, by Kernigham and Ritchie, defines a standard language and runtime library and also discusses differences in several minicomputer implementations. The book, however, does not define a standard for the layout and presentation of C code, considering this a matter of personal taste. After reading a large number of C programs from many sources, I have decided that programmers pay too little attention to this aspect of C programming. In the following paragraphs, I will explain the problem as I see it, suggest a standard for C code layout, and propose a possible solution.

Whenever I receive a new piece of C code, I always check to see how the programmar has presented the code. A clear presentation with many comments and an uncluttered look is important for maintaining such code and for aiding other programmers who must understand it. More often than not, the code looks something like the following:

```
main( ){printf("Hello world\n");
}
```

In this case the programmer hasn't formatted the code properly. This makes it difficult to follow the inherent block structure of the language The cluttered look that results from improper indentation is often accompanied by a paucity of comments. The result is code that is hard to understand, improve, or debug.

The style presented in Kernigham and Ritchie is consistent but not optimal, since it does not present blocks and block nesting in the clearest way possible. For example, an if . . . else loop would appear as in Figure 1 (page 17). Using the same fragment, my preference is for the format shown in Figure 2 (page 17). In this format, eight-space (standard) tabs are used for indentation (as opposed to the six-space indentation used by the book). Braces are almost always on their own lines, and the lowest level braces appear at the left margin. Braces indicate the nesting level of the expression that invoked them, and their contents themselves are indented an additional level. Only one statement is placed on a line, and comments are added liberally to make the code more readable. Finally, the space used by Kernigham and Ritchie between keywords and their parenthetical expressions is omitted.

This example illustrates the layout standard I am proposing, which is summarized in Table I (page 18). My purpose in proposing this standard is to induce C programmers to think more carefully about the layout and presentation of their C code. Clearly, there is more to programming than the layout of the code. The data structures used and the program structure are crucial in producing a good piece of software. However, without good layout the best program may be difficult to understand, maintain, or improve. I invite reader response to the layout proposal.

A Possible Solution

I expect that even if *The C Programming Language* described a layout standard, some programmers would deviate from it. In order to provide programmers with flexible code layouts, C beautifiers can be created. Such programs take C code as input and, by adding/removing white space characters, reformat the C code to some individual layout specification. This allows each programmer to distribute code in a standard layout, while using his or her individual layout for local copies of the code.

Beautifiers already exist. For example, Berkeley UNIX has a beautifier called CB. This program is fairly simple-minded but will convert unformatted C code that uses a minimum amount of white space into the Kernigham and Ritchie-type layout. More ambitious beautifiers can be created; readers who create their own beautifiers are invited to submit their code for publication in this column.

Runtime Libraries

The C Programming Language defines a standard runtime library for C. Some of the features provided are only feasible under UNIX. Thus, for CP/M, CP/M-86, and MS-DOS implementations, only part of the runtime library can be supported. Yet some C compilers do not provide a compatible subset library. The result is code that cannot be easily transported from compiler to compiler or from machine to machine. Such compilers negate one of the primary purposes of the C programming language: portability.

The BDS C compiler is one such software product. It is an excellent subset compiler, but its runtime library is incompatible with the standard. After using BDS C for more than three years, I have accumulated a significant number of useful subroutines and programs. Unfortunately, some of this software depends heavily on the BDS C runtime library. This code will require a great deal of work before it can be used with another compiler. Be aware of this pitfall and be prepared to live with the consequences if you choose a compiler with a nonstandard runtime library.

BDS C is not the only compiler whose runtime library is nonstandard. The Whitesmith C compilers are another. Since Whitesmith compilers are available for many different environments, however, portability between Whitesmith compilers is immediate. Nevertheless, my preference is for code written for use with the standard library and for compilers that support that library in full or subset form.

Link Formats

Besides incompatible runtime libraries, there is the question of subroutine linkage and linkage editor formats. Once again BDS C is nonstandard; that is, it uses its own linker instead of conforming to the Microsoft .REL format. (Some 8080/Z80 compilers do support the standard, such as the Q/C compiler from The Code Works.) Compatibility with the .REL format is a considerable blessing if you plan to link other software to your C programs.

The lack of linkage compatibility is not limited to the CP/M-80 world. A wide variety of C compilers sold for MS-DOS and CP/M-86 also fail to use the appropriate standard. Most notably, Computer Innovations' excellent C86 compiler uses its own internal format, which makes it impossible to use C86 to produce subroutines for other compiled languages under MS-DOS or CP/M-86. Fortunately, the MANX AZTEC compilers can be used as an option to produce Microsoft format object code.

The reasons for linkage incompatibility are probably manifold. The usual response from the companies producing the incompatible products is that they prefer their own format to Microsoft's. Why is this? I suppose they assume that programmers don't care about standards or don't think that their end users would find a compatible linkage format useful. In any case, be aware of the linkage format used by the C compilers you select.

Conclusion

This column is intended to be a forum for reader discussion as much as it is a resource. This first installment has examined a potential C code layout standard, and has discussed incompatibility in C object code formats and runtime libraries. Discussion and comment on these and other topics are welcome. Questions, useful programs, problems, insights, and suggestions are all encouraged. I look forward to hearing from you.

Next time, we'll begin discussing how UNIX-like environments tend to produce noninteractive code. . . .

DDJ

(Table I begins on page 18)

```
if ((fp = fopen(argv[1],"r")) == NULL) { /* not found */
        fprintf(stderr,"%s not found\n",argv[1]);
        exit(1);
} else {                        /* the file is present */
        fprintf(stderr,"%s is on-line\n",argv[1]);
        fclose(fp);
}
```

Figure 1.

```
if((fp = fopen(argv[1],"r")) == NULL)   /* not found */
{
        fprintf(stderr,"%s not found\N",argv[1]);
        exit(1);  /* exit with error status 1 */
}
else    /* the file is present */
{
        fprintf(stderr,"%s is on-line\n",argv[1]);
        fclose(fp);  /* close the file */
}
```

Figure 2.

Table I
Proposed Standard for C Code Layout

1. Standard tabs are used for program indentation.
   a. Four-space tabs may be used, if desired, after column 56. This is permitted in order to prevent line wrap on 80-column displays.
2. A brace is generally on a line by itself.
   a. The lowest level braces appear on the left margin.
   b. Comments may appear on lines containing a brace, but only after the brace.
   c. Braces are indented to the same depth as the statement that invoked the block they frame.
   d. In variable initializations, the opening brace may be placed on the same line as the equal sign, but the closing brace must have the same indentation as the opening one.
   e. In general, no statements may appear on the same line as a brace.
3. A block begins with a left brace and ends with a right brace. Its contents are indented an extra level to indicate the nesting depth.
   a. Whenever a block is longer than 24 lines (a standard CRT page), a comment should follow the closing brace to indicate the block that the brace closes.
      i. This applies to whole functions as well as regular blocks.
      ii. This rule should also be applied to shorter blocks when block nesting makes the code complex and these comments improve the readability.
   b. The opening and closing braces of a block are always indented identically.
   c. The case/default labels of a switch statement are always indented a level, like statements in a block. The statements that follow these labels are always indented an extra level to improve readability.
   d. Regular labels (destinations for GOTO's) are always placed at the left margin, regardless of nesting depth.
   e. When a null block is used (e.g., "{}"), it may appear on the same line as statements (e.g., do {} while(expr);).
   f. If a single statement is used instead of a block, it is indented a single level, just as if it were surrounded by braces.
      i. Null statements (i.e., just a ";") are indented in the same way as regular statements.
4. White space is added in expressions and assignments to improve readability.
   a. Relational operators are delimited by single spaces.
   b. Equal signs are delimited by single spaces.
   c. Unary operators are not separated by space from their operands.
   d. Parentheses are added to improve readability in complex expressions, even if they are not required to produce correct evaluation.
      i. A return statement always has a set of parentheses surrounding its expression.
   e. No white space character is placed between function names and their parenthesized argument list.
   f. No white space character is placed between a keyword (e.g., if) and its parenthesized argument.
5. Comments are added liberally to make the program read easily.
   a. If a comment requires more than one line, the start/end comment tokens are placed on lines containing no comment text. In this case, the start/end tokens are indented identically.
   b. If a comment fits on a single line, the start/end tokens must also be placed on that line.
6. Variables are always declared, even if your version of C has a default type. Always explain the purpose of your variables.
   a. Declare the variables in logical groups and include a comment on the same line as the declaration to describe the function(s) of the variable(s).
   b. Avoid numerous declarations on a single line.
   c. Explain complex pointer declarations.
   d. Variable names are always lower case only.
   e. External variable declarations may be indented a single level for greater readability.
7. Constants created with #define are always upper case. Macros created with #define may be upper or lower case.
   a. As with variables, constants should have comments to explain their purpose.
   b. Macros should be explained via comments to avoid misunderstandings about their uses. This is especially important since macros tend to be cryptic.
   c. Restrict #define statements to the beginning of code files (i.e., before the first function). This avoids the potential for redefinition and other confusion.
8. Other items.
   a. The else . . if(expr) construct is placed on a single line as if it were a single keyword.
   b. Function names are always lower case.
9. Portability considerations.
   a. When using a nonstandard C feature, always prepare a portable alternative that may be selected via conditional compilation.
   b. Indicate any subtle uses of sign extension and type conversions made by your program or your specific compiler.
   c. Indicate any deviations in the way your compiler handles pointer arithmetic.
   d. Include the ampersand operator ("&") when you pass pointers to structures, even if your compiler doesn't require it. Someone else's compiler may support passing structures.
   e. Indicate any deviations of your runtime library from the standard.
   f. Indicate any deviation in the (argc,argv) command line conventions made by your compiler.

UNIX to CP/M
Floppy Disk File Conversion

by W. E. Wilson

W. E. Wilson, Nuclear Radiation Center, Washington State University, Pullman, WA 99164.

UNIX is a trademark of Western Electric Corporation – CP/M is a trademark of Digital Research – Z80 is a trademark of Zilog.

The increasing availability of good, moderate-cost C compilers for use on microprocessor systems has generated significant interest in the C language among microprocessor system users. Ron Cain's Small-C compiler with its improvements (see J. E. Hendrix's Small C Compiler, v. 2, *DDJ* Nos. 74, 75) and the BDS C compiler have spawned a large C users group among CP/M users. The C language was specifically developed for the UNIX operating system, and there is a large amount of software written in C available on UNIX systems. Since the file structures of UNIX and CP/M files are worlds apart, there is no direct way to read UNIX C source files under CP/M. The purpose of this article is to describe the structure of UNIX files and a program written by the author to read UNIX floppy disks under CP/M.

Examining UNIX Files Under CP/M

The Bell Labs UNIX programming manual describes the basic structure of the UNIX file system. The detailed information listed in the next section was actually obtained by comparing UNIX floppy disk dumps with the information in the UNIX manuals. A program provided with the ZMAC macro assembler called DUMPDSK significantly simplified the process and was also used as a pattern for writing UNIXTCPM. DUMPDSK is an example program on the use of ZMAC and the associated ZSUBLIB subroutine library. DUMPDSK dumps a SSSD 8-inch IBM 3740 format floppy disk starting at track 0, sector 1, using an interleave factor of 6. (Most DEC floppy disk drive systems used with UNIX use the IBM 3740 format and an interleave factor of 6.) The UNIXTCPM program which accompanies this article (Listing One, page 24) will only work with UNIX files written by such drives.

UNIX Files

The UNIX operating system provides a hierarchical file system, which simply means that file directories and files are organized in a tree structure. The structure is really quite simple and all files, including directories, are viewed simply as a one-dimensional array of bytes. A disk is viewed as a conically random access device with data partitioned into 512-byte blocks. Thus the basic unit of disk storage is a block of 512 bytes.

Blocks are numbered sequentially starting at block 0, which comprises the first 512 bytes on a disk. Block 0 is not used by the file system and is reserved for system booting operations. The second block, or block 1, is called the "super block." This block contains the capacity of the file device, available free blocks, and other information as described in Table I (page 22).

The next block in the UNIX file system, block 2, is the beginning of a structure called "I-list." I-list is a sequence of structures that relates a given file to the blocks comprising the file on the disk. Each successive 64 bytes in I-list holds a file allocation descriptor, one per file, called an "inode." The inode is the basic and most important file structure element in the UNIX file system. The structure of each inode is described in Table II (page 22).

The UNIX file system device format just described is depicted in Figure 1 (page 23). I-list runs from block 2 to isize, which is the first two bytes in the super block. Isize thus designates the start of the data area and the end of I-list. Each inode is assigned a number called the I-number. Thus, I-number 1 designates inode 1, or the first 64 bytes of I-list; I-number 2 designates the second 64 bytes of I-list, etc. By convention, I-number 2 is the inode for the root directory blocks in the file system and I-number 1 is, for all practical purposes, unused.

Directory files in UNIX are exactly like ordinary files except that they cannot be written into by a user. The mode byte in the inode of a file indicates the type of file involved. Each directory block has space for 32 16-byte directory entries. The first and second bytes of each entry are the I-number for the file. The remaining 14 bytes are the name of the file left adjusted in the field. The first directory entry in the root directory is for the root directory block itself and the second is for the parent directory. The third entry is for the first actual file. The UNIX file system just described is shown in Figure 2 (page 23).

Accessing a file on a UNIX disk may be accomplished by the following sequence of operations:

(1) Read address 1 of inode number 2 to obtain the block number of the root directory. In the case of a floppy disk with an IBM 3740 format, this would be CP/M track 0, sector 23, bytes 4C, 4D, and 4F. If the root directory is filled, addr 2 of inode 2 points to the next directory block, etc.

(2) Read the root directory block designated above. The root directory on a floppy disk generally is in block 24, starting in CP/M track 3, sector 6.

(3) Locate the desired file name in the directory.

(4) Pick up the I-number for the file from the first two bytes of the directory entry.

(5) Locate the inode for the file. I-number 1 corresponds to the first 64 bytes of CP/M track 0, sector 23, I-number 2 to the second 64 bytes. I-number 3 corresponds to the first half of CP/M track 0, sector 3, etc.

(6) Read the file blocks assigned to the file starting in inode byte C. Each block address is three bytes long and the first ten blocks are for directly addressed file blocks. The last three addr values are for indirect pointers, as shown in Figure 3 (page 23).

(7) Read the blocks assigned to the file. In the event the file consists of more than ten blocks, address 11 is the block number of an array of pointers to the next 128 blocks in the file. Each block number is four bytes long in the pointer block.

UNIXTCPM Program Design

The UNIXTCPM program performs the following listed set of operations:

(1) Command Line Input

```
UNIXTCPM    filename.ext  $081$
Filename ------------^
I-number for file ------------^
```

(2) Determine Track and Sector of Inode

SQSECT = Sequential Sector Number = (I-number 1)/2 + 8

TRACK = Integer portion of SQSECT/26

SCTRANN = Position of sector in Sector Translation Table

SCTRANN = Remainder of SQSECT/26

(3) Read Inode into INDBUF

If the I-number is odd, inode is in the first half of the sector; if even, then it is in the second half.

(4) Check inode 11 at location 2C, 2D, and 2E for indirect block. If address 11 is not zero, the program reads the pointer block specified by address 11 into INDBUFX. This program does not consider second order indirection, and thus is limited to files of 70K bytes or less.

(5) Read a specified file from the UNIX disk in unit B and write a CP/M file on unit A.

UNIXTCPM Program Source Listing

The UNIXTCPM program was written in Z80 assembly language using the ZMAC macro assembler and the ZSUBLIB subroutine library provided with this product. The resultant REL file was then linked to form a COM file using ZLINK. The program was tested using three different floppy disks with a different set of files on each. The files were created using UNIX version 7 and UNIXTCPM was able to read all of them. The source code file is given in Listing One on page 24. Included with the unixtcpm program is NLFILTER.COM (Listing Two, page 37), which changes the UNIX newline to a CR-LF in the CP/M file.

DDJ

(Listings begin on page 24)

Field Name	Bytes	Function
isize	2	Block Num at End inode + 1 = First Data Block
fsize	4	Size of entire disk in blocks
nfree	2	Pointer to last allocated block in ifree array
free	4 x 50	Array of 50 pointers to free blocks
ninode	2	Number of free Inumbers to inode array
inode	2 x 100 + 2	Array of 100 pointers to inodes in I-list
flock	1	Lock during free list manipulation
ilock	1	Lock during I-list manipulation
fmod	1	Super block mod flag
ronly	1	Read only flag
time	4	Time super block last mod
tfree	2	Total free inodes
m	2	Interleave factor 1
n	2	Interleave factor 2
fname	6	File system name
fpack	6	File system pack name

Table I
Super Block Structure

Field Name	Bytes	Function
mode	2	Type of file
nlink	2	Number of Dir entries for this inode
uid	2	Owner I.D.
gid	2	Group I.D.
size	4	Number of bytes in file
addr	13 x 3 + 1	Array of 13 3-byte block numbers assigned to file, first 10 direct, last 3 indirect
atime	4	Time last accessed
mtime	4	Time last modified
ctime	4	Time created

Table II
Inode Structure

Circle no. 32 on reader service card.

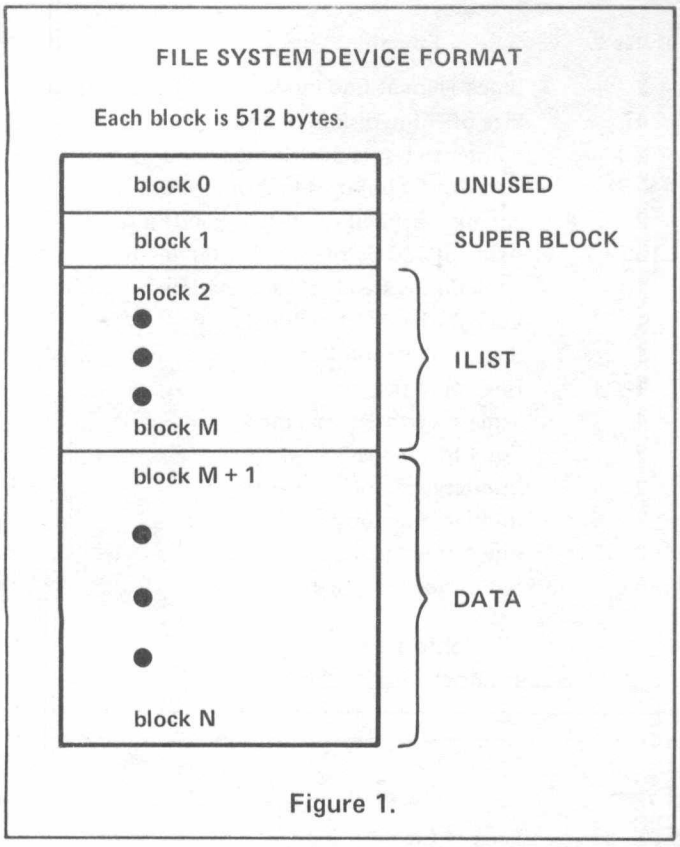


Figure 1.

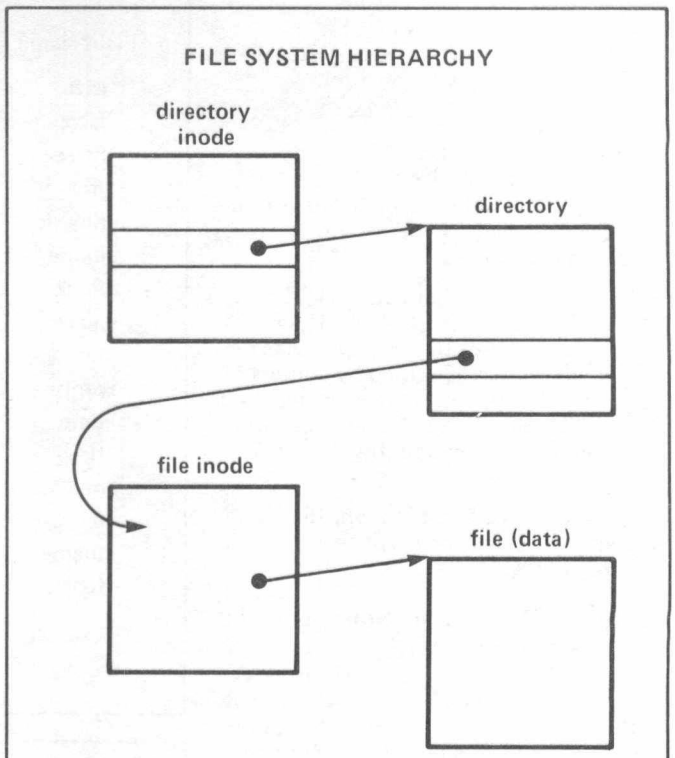


Figure 2.

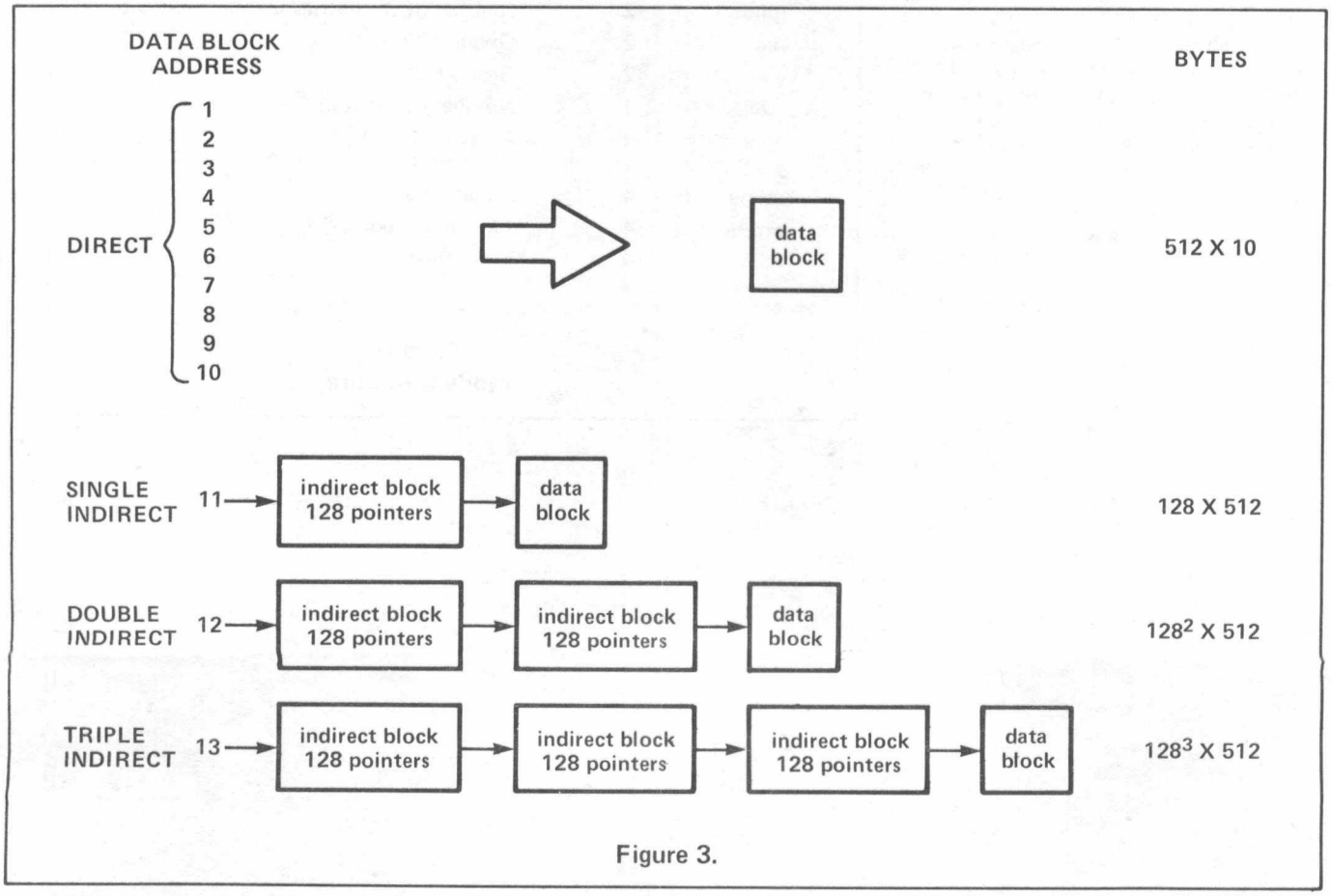


Figure 3.

UNIX to CP/M (Text begins on page 20)

Listing One

```
                        0001  ;*******************************************************
                        0002  ;  UNIXTCPM  **UNIX FILE TO CP/M FILE TRANSFER**
                        0003  ;        This program assumes that UNIXTCPM.COM is
                        0004  ;        on disk unit A and that the UNIX disk is
                        0005  ;        is in dive unit B.  The CP/M equivolent of
                        0006  ;        the specified UNIX file will be on unit A.
                        0007  ;        UNIXTCPM uses the ZSUBLIB routine GETSECT
                        0008  ;        that employs direct BIOS calls to read a
                        0009  ;        single sector.  This program assumes an
                        0010  ;        interleave factor of 6 for the UNIX disk.
                        0011  ;
                        0012  ;  Command Line=UNIXTCPM filename.ext $XXX$<RET>
                        0013  ;                  where xxx = UNIX file I number.
                        0014  ;
                        0015  ;        Use DUMPDSK on the UNIX floppy disk to
                        0016  ;        determine the I number for the UNIX file
                        0017  ;        to be transfered to CP/M.
                        0018  ;
                        0019  ;        The message "Fatal Error" will be displayed
                        0020  ;        and UNIXTCPM will abort if a bad I number
                        0021  ;        or bad track and sector numbers a produced.
                        0022  ;
                        0023  ;        UNIX newline in CP/M file must be changed
                        0024  ;        to CR LF using program NLFILTER.COM
                        0025  ;*******************************************************
                        0026  ;
                        0027  ;        --ZSUBLIB ROUTINES USED BY UNIXTCPM--
                        0028          EXT     FCB01,FNTFCB,CREATE,CLOSE,DIV16
                        0029          EXT     OPEN,SETBUFF,CONCRLF,ADCTBIN
                        0030          EXT     GETSECT,SELECT,PRNTFN,PUTMC
                        0031          EXT     WRTBUF1,SCTBUF1
                        0032  ;
                        0033          REL                     ;SET PROGRAM RELATIVE
                        0034  ;
     (0100)             0035          ORG     0100H           ;NORMAL PGM ORG
0100 C36804             0036  BEGIN   JP      START
                        0037  ;
                        0038  ;-------------------------------------------------------
                        0039  ;--STACK, MESSAGES, BUFFERS, AND PARAMETER STORAGE--
                        0040  ;-------------------------------------------------------
                        0041  ;--RESERVE STACK SPACE--
0103 (0050)             0042          DEFS    80
                        0043  STACK:
                        0044  ;
                        0045  ;--SIGNON MESSAGE
0153 0D0A               0046  SIGNON  DEFB    0DH,0AH
0155 3E3E2055           0047          DEFB    '>> UNIX to CP/M '
     4E495820
     746F2043
     502F4D20
0165 46696C65           0048          DEFB    'File Transfer <<'
     20547261
     6E736665
     72203C3C
0175 0D0A24             0049          DEFB    0DH,0AH,'$'
                        0050  ;--EXIT MESSAGE--
0178 0D0A               0051  EXTMSG  DEFB    0DH,0AH
017A 2A2A2045           0052          DEFB    '** End of Trans'
     6E64206F
```

(Continued on page 26)

UNIX to CP/M (Listing continued, text begins on page 20)

Listing One

```
      66205472
      616E73
0189  66657220     0053          DEFB    'fer **',0DH,0AH
      2A2A0D0A
0191  24           0054          DEFB    '$'
                   0055  ;--FILE NAME MESSAGE--
0192  0D0A         0056  MSG1    DEFB    0DH,0AH
0194  54686520     0057          DEFB    'The File beins '
      46696C65
      20626569
      6E6720
01A3  7472616E     0058          DEFB    'transfered is  '
      73666572
      65642069
      732020
01B2  24           0059          DEFB    '$'
                   0060  ;--ERROR MESSAGE-
01B3  0D0A         0061  MSG2    DEFB    0DH,0AH
01B5  46617461     0062          DEFB    'Fatal Error'
      6C204572
      726F72
01C0  0D0A24       0063          DEFB    0DH,0AH,'$'
                   0064  ;
                   0065  ;--BINARY I NUMBER
01C3  0000         0066  INUM    DEFB    00H,00H          ;EXTRA 0 FOR LD HL(INUM
                   0067  ;
                   0068  ;--THREE DIGIT ASCII UNIX FILE I number--
01C5  00           0069  DIG1    DEFB    00H
01C6  00           0070  DIG2    DEFB    00H
01C7  00           0071  DIG3    DEFB    00H
01C8  00           0072  DEND    DEFB    00H
                   0073  ;
                   0074  ;--POINTER TO POSITION IN SECTRAN--
01C9  00           0075  SCTPTR  DEFB    00H
                   0076  ;
                   0077  ;--SECTOR TRANSLATION TABLE--
01CA  01070D13     0078  SECTRAN DEFB    01,07,013,019,025,05,011
      19050B
01D1  11170309     0079          DEFB    017,023,03,09,015,021,02
      0F1502
01D8  080E141A     0080          DEFB    08,014,020,026,06,012,018
      060C12
01DF  18040A10     0081          DEFB    024,04,010,016,022,00
      1600
                   0082  ;
                   0083  ;--TRACK NUMBER--
01E5  00           0084  TRACK   DEFB    00H
                   0085  ;--SECTOR NUMBER--
01E6  00           0086  SECT    DEFB    00H
                   0087  ;
                   0088  ;--INDIRECT POINTER BLOCK SWITCH--
01E7  00           0089  INDSW   DEFB    00H
                   0090  ;
                   0091  ;--BUFFER FOR inode--
01E8  (0080)       0092  INDBUF  DEFS    0128             ;ONE SECTOR
                   0093  ;
                   0094  ;--BUFFER FOR 1ST INDIRECT POINTER BLOCK
0268  (0200)       0095  INDBUFX DEFS    0512             ;4 SECTORS
                   0096  ;
```

(Continued on page 28)

```
                        0097  ;-----------------------------------------;
                        0098  ;  ** ACTUAL PROGRAM STARTING POINT **    ;
                        0099  ;-----------------------------------------;
                        0100  ;
0468  315301              0101  START   LD      SP,STACK          ;SET STACK POINTER
                        0102  ;--DO SIGNON--
046B  115301              0103          LD      DE,SIGNON
046E  CD0000#             0104          CALL    PUTMC
                        0105  ;--SELECT DRIVE A--
0471  AF                  0106          XOR     A
0472  CD0000#             0107          CALL    SELECT
                        0108  ;--USE FNTFCB TO GET FILE NAME FROM COMMAND LINE--
0475  110000#             0109          LD      DE,FCB01
0478  215C00              0110          LD      HL,05CH           ;COMMAND LINE
047B  CD0000#             0111          CALL    FNTFCB
                        0112  ;--PRINT FILE NAME INFO--
047E  119201              0113          LD      DE,MSG1
0481  CD0000#             0114          CALL    PUTMC
0484  110000#             0115          LD      DE,FCB01
0487  CD0000#             0116          CALL    PRNTFN
048A  CD0000#             0117          CALL    CONCRLF           ;CON NEWLINE
                        0118  ;--CREATE CP/M FILE--
                        0119  ;--NOTE: NO CHECK IS MADE FOR DUPLICATE
                        0120  ;         FILE BY THIS NAME ON UNIT A--
048D  110000#             0121          LD      DE,FCB01
0490  CD0000#             0122          CALL    CREATE
                        0123  ;--GET I NUMBER FROM COMMAND LINE--
0493  CD2605              0124          CALL    GETINUM
                        0125  ;--CONVERT I NUM TO INODE TRACK AND SCTT--
0496  CD5E05              0126          CALL    CNVINUM
                        0127  ;--GET UNIX FILE inode from disk B--
0499  11E801              0128          LD      DE,INDBUF
                        0129  ;--READ inode SECTOR INTO INDBUF--
                        0130  ;--Note: CNVINUM sets track and sector--
049C  CD8405              0131          CALL    RD1SECT           ;READ ONE SECTOR
                        0132  ;--CHECK addr 11 for 1st INDIRECT BLOCK--
049F  3AC301              0133          LD      A,(INUM)          ;GET I number
04A2  E601                0134          AND     01H               ;CK for odd or even
04A4  2A1302              0135          LD      HL,(INDBUF+02BH)          ;addr 11 odd
04A7  ED5B5302            0136          LD      DE,(INDBUF+06BH)          ;addr 11 even
04AB  2001                0137          JR      NZ,USEFHLF        ;odd, use 1st half
04AD  EB                  0138          EX      DE,HL             ;even, use 2nd half
04AE  7C                  0139  USEFHLF LD      A,H
04AF  B5                  0140          OR      L
04B0  2806                0141          JR      Z,NOINDP          ;NO INDIRECT PTRS
                        0142  ;--GET FIRST ORDER INDIRECT POINTER BLOCK--
                        0143  ;--NOTE: HL contains addr 11 = pointer block num--
04B2  CD8005              0144          CALL    CNVBLK            ;BLK TO T and S
04B5  CDC805              0145          CALL    RDINDPB           ;Read ptr block
04B8  DD21F501            0146  NOINDP  LD      IX,INDBUF+0DH     ;SET BLOCK INDEX in ino

04BC  3AC301              0147          LD      A,(INUM)          ;GET I number
04BF  E601                0148          AND     01                ;CK for odd or even
04C1  2005                0149          JR      NZ,ODDINM
                        0150  ;--EVEN I number, Use 2nd half inode BLOCK--
04C3  114000              0151          LD      DE,040H           ;ADD 64
04C6  DD19                0152          ADD     IX,DE
                        0153  ;---------------------------------;
                        0154  ;       **MAIN PROGRAM LOOPS**    ;
                        0155  ;---------------------------------;
```

(Continued on next page)

UNIX to CP/M (Listing continued, text begins on page 20)

Listing One

```
                    0156  ;--READ 10 DIRECT ADDR BLOCKS LOOP--
04C8  060A          0157  ODDINM  LD      B,010           ;LOOP COUNT
04CA  C5            0158  RDNXBK  PUSH    BC
                    0159  ;--READ NEXT BLOCK--
04CB  CDF704        0160          CALL    RDBLOCK
04CE  C1            0161          POP     BC
                    0162  ;--END RDNXBK LOOP--
04CF  10F9          0163          DJNZ    RDNXBK
                    0164  ;--END OF DIRECTLY ADDRESSED BLOCKS--
                    0165  ;-------------------------------------------------;
                    0166  ;   USE 1ST INDIRECT POINTER BLOCK                ;
                    0167  ;   READ MAX OF 128 BLOCKS 1ST INDIRECTION        ;
                    0168  ;-------------------------------------------------;
04D1  DD216A02      0169          LD      IX,INDBUFX+2    ;SET BLOCK INDEX
04D5  3EFF          0170          LD      A,0FFH
04D7  32E701        0171          LD      (INDSW),A       ;SET INDSW
04DA  0680          0172          LD      B,0128          ;LOOP COUNT
                    0173  ;--READ NEXT INDIRECT BLOCK--
04DC  C5            0174  RDNXIBK PUSH    BC
04DD  CDF704        0175          CALL    RDBLOCK
04E0  C1            0176          POP     BC
04E1  10F9          0177          DJNZ    RDNXIBK
                    0178  ;--QUIT AT END 1ST INDIRECT BLOCK
                    0179  ;  FILE SIZE EXCEEDS 70K bytes--
                    0180  ;
                    0181  ;---DONE, CLOSE FILES--
04E3  110000#       0182  DONE    LD      DE,FCB01
04E6  CD0000#       0183          CALL    CLOSE
                    0184  ;--SEND EXIT MSG--
04E9  117801        0185          LD      DE,EXTMSG
04EC  CD0000#       0186  ERXIT   CALL    PUTMC
04EF  C30000        0187          JP      00H             ;BACK TO CP/M
                    0188  ;--FATAL ERROR--
04F2  11B301        0189  ERROR   LD      DE,MSG2
04F5  18F5          0190          JR      ERXIT
                    0191  ;---------------------------------;
                    0192  ;       **END MAIN PROGRAM**      ;
                    0193  ;         SUBROUTINES BELOW       ;
                    0194  ;---------------------------------;
                    0195  ;
                    0196  ;--READ ONE BLOCK--
                    0197  ;
                    0198  RDBLOCK:
                    0199  ;--GET BLOCK NUM FROM inode addr or PTR block--
04F7  DD6E00        0200          LD      L,(IX+0)
04FA  DD6601        0201          LD      H,(IX+1)
                    0202  ;--CHECK FOR NULL addr--
04FD  7D            0203          LD      A,L
04FE  B4            0204          OR      H
04FF  28E2          0205          JR      Z,DONE          ;END FILE
                    0206  ;--INDEX TO NEXT addr--
0501  DD23          0207          INC     IX
0503  DD23          0208          INC     IX
0505  DD23          0209          INC     IX
0507  3AE701        0210          LD      A,(INDSW)
050A  B7            0211          OR      A
050B  2802          0212          JR      Z,NTINDRC
```

```
050D  DD23          0213          INC     IX              ;INDIRECT 4 BYTES
                    0214  ;--SET TRK AND SSTN FROM BLK NUM--
050F  CD8005        0215  NTINDRC CALL    CNVRLK
                    0216  ;--READ one BLK (4 SECT) LOOP--
0512  0604          0217          LD      B,04
0514  C5            0218  RD4SLP  PUSH    BC
                    0219  ;--USE SCTBUF1 IN WRTBUF1 ROUTINE--
0515  110000#       0220          LD      DE,SCTBUF1      ;SECTOR BUFFER
0518  CD8405        0221          CALL    RD1SECT         ;READ 1 SECTOR
                    0222  ;--SELECT DISK FOR WRITE
051B  AF            0223          XOR     A               ;UNIT A
051C  CD0000#       0224          CALL    SELECT
                    0225  ;--WRITE OUT SCTBUF1 USING FCB01--
051F  CD0000#       0226          CALL    WRTBUF1
0522  C1            0227          POP     BC
                    0228  ;--END RD4SLP LOOP--
0523  10EF          0229          DJNZ    RD4SLP
0525  C9            0230          RET
                    0231  ;
                    0232  ;---GET UNIX FILE I NUMBER FROM COMMAND LINE--
                    0233  ;
                    0234  ;--THREE DIGITS FOR I NUMBER--
0526  216C00        0235  GETINUM LD      HL,06CH
0529  CD4B05        0236          CALL    GETDIG          ;GET DIGIT
052C  32C501        0237          LD      (DIG1),A
052F  CD4B05        0238          CALL    GETDIG
0532  32C601        0239          LD      (DIG2),A
0535  CD4B05        0240          CALL    GETDIG
0538  32C701        0241          LD      (DIG3),A
053B  11C501        0242          LD      DE,DIG1         ;STRING PTR TO DE
                    0243  ;--CONVERT ASCII DEC STRING TO BIN NUM--
053E  CD0000#       0244          CALL    ADCTBIN
                    0245  ;--BINARY I NUM IN HL, MAX VALUE=160--
0541  7D            0246          LD      A,L             ;SAVE LSB ONLY
0542  FEA0          0247          CP      0160
0544  D2F204    R   0248          JP      NC,ERROR        ;ILLEGAL I NUM
0547  32C301        0249          LD      (INUM),A        ;SAVE I NUMBER
054A  C9            0250          RET
                    0251  ;
                    0252  ;---GET DIGIT FROM COMMAND LINE--
                    0253  ;
054B  7E            0254  GETDIG  LD      A,(HL)
054C  23            0255          INC     HL
054D  FE20          0256          CP      ' '             ;SKIP BLANKS
054F  28FA          0257          JR      Z,GETDIG
0551  FE24          0258          CP      '$'             ;SKIP $
0553  28F6          0259          JR      Z,GETDIG
0555  FE00          0260          CP      00
0557  28F2          0261          JR      Z,GETDIG        ;SKIP ZERO
0559  FE3A          0262          CP      03AH            ;SKIP NON NUM
055B  30EE          0263          JR      NC,GETDIG
055D  C9            0264          RET
                    0265  ;
                    0266  ;--CONVERT I NUMBER TO INODE TRK AND SCTT--
                    0267  ;
055E  2AC301        0268  CNVINUM LD      HL,(INUM)       ;GET I NUMBER
0561  2B            0269          DEC     HL
0562  110200        0270          LD      DE,02
```

(Continued on next page)

Listing One

```
0565  CD0000#    0271          CALL    DIV16           ;(I NUM-1)/2
0568  110800      0272          LD      DE,08H
056B  19          0273          ADD     HL,DE           ;(I NUM -1)/2 +8
                  0274  ;
                  0275  ;--CONVERT SEQUENTIAL SECT NUM IN HL
                  0276  ;    TO TRACK AND SECT-TRAN NUMBER--
                  0277  ;
056C  111A00      0278  CNVSSNM LD      DE,026
056F  CD0000#     0279          CALL    DIV16
                  0280  ;--TRACK = Intiger portion (SSN/26)--
0572  7D          0281          LD      A,L
0573  FE4D        0282          CP      077
0575  D2F204      0283          JP      NC,ERROR        ;ILLEGAL TRACK NUM
0578  32E501      0284          LD      (TRACK),A       ;SAVE TRACK
                  0285  ;--Relative sector (secttran num) = remainder(SSN/26)--
057B  7B          0286          LD      A,E
057C  32C901      0287          LD      (SCTPTR),A      ;SAVE SECT-TRAN
057F  C9          0288          RET
                  0289  ;
                  0290  ;--CONVERT BLOCK NUM IN HL TO TRACK AND SECT--
                  0291  ;
0580  29          0292  CNVBLK  ADD     HL,HL           ;x 2
0581  29          0293          ADD     HL,HL           ;x 4
0582  18E8        0294          JR      CNVSSNM
                  0295  ;
                  0296  ;--READ ONE UNIX FILE SECTOR INTO DMA SPCIFIED
                  0297  ;     IN DE AND INC TO NEXT SEQUENTIAL SECTOR--
                  0298  ;

0584  D5          0299  RD1SECT PUSH    DE
0585  3E01        0300          LD      A,01
0587  CD0000#     0301          CALL    SELECT          ;SELECT UNIT B
058A  D1          0302          POP     DE
058B  CD0000#     0303          CALL    SETBUFF         ;SET DMA
                  0304  ;--CONVERT SECTRAN POINTER TO SECTOR--
058E  3AC901      0305          LD      A,(SCTPTR)
0591  21CA01      0306          LD      HL,SECTRAN      ;BASE TRANS TAB
0594  85          0307          ADD     A,L             ;ADD OFFSET
0595  6F          0308          LD      L,A
0596  3001        0309          JR      NC,NOCARY
0598  23          0310          INC     HL
                  0311  ;--GET SECTOR  NUM FROM SECTRAN TABLE--
0599  7E          0312  NOCARY  LD      A,(HL)
059A  32E601      0313          LD      (SECT),A        ;SAVE SECT
059D  FE00        0314          CP      00H
059F  CAF204      0315          JP      7,ERROR         ;ILLEGAL SECT NUM
05A2  FE1B        0316          CP      27
05A4  D2F204      0317          JP      NC,ERROR        ;ILLEGAL SECT NUM
                  0318  ;--LOAD SECTOR NUM INTO C--
05A7  4F          0319          LD      C,A             ;SECT FOR GETSECT
                  0320  ;--LOAD TRACK NUM INTO B--
05A8  3AE501      0321          LD      A,(TRACK)
05AB  47          0322          LD      B,A             ;TRACK FOR GETSECT
                  0323  ;--DIRECT BIOS READ ONE SECTOR --
05AC  CD0000#     0324          CALL    GETSECT         ;READ SECT INTO SCTBUF1
05AF  DAF204      0325          JP      C,ERROR         ;BAD TRK/SECT INFO
                  0326  ;--INC SECTOR NUMBER FOR NEXT READ--
05B2  3AC901      0327          LD      A,(SCTPTR)      ;GET TAB OFFSET
05B5  FE19        0328          CP      025             ;END ?
05B7  200A        0329          JR      NZ,NOTEND
```

```
                        0330  ;--END TRACK, INC TRACK AND ZERO SCTPTR--
05B9  3AE501            0331          LD      A,(TRACK)
05BC  3C                0332          INC     A
05BD  32E501            0333          LD      (TRACK),A
05C0  AF                0334          XOR     A
05C1  1801              0335          JR      ENDTK
05C3  3C                0336  NOTEND  INC     A
05C4  32C901            0337  ENDTK   LD      (SCTPTR),A      ;NXT SECT
05C7  C9                0338          RET
                        0339  ;
                        0340  ;--READ 1ST ORDER INDIRECT POINTER BLOCK--
                        0341  ;
05C8  116802            0342  RDINDPB LD      DE,INDBUFX
05CB  CD8405            0343          CALL    RD1SECT
05CE  11E802            0344          LD      DE,INDBUFX+128
05D1  CD8405            0345          CALL    RD1SECT
05D4  116803            0346          LD      DE,INDBUFX+256
05D7  CD8405            0347          CALL    RD1SECT
05DA  11E803            0348          LD      DE,INDBUFX+384
05DD  CD8405            0349          CALL    RD1SECT
05E0  C9                0350          RET
                        0351  ;---END OF PROGRAM--
05E1  (0000)            0352          END

Total 2nd Pass Errors   =          0
JPs that could be JRs   =          1
Z80/8080 Parity Usage   =          0


** Symbol  Table **

ADCTBIN  053F#     BEGIN    0100     CLOSE    04E7#     CNVBLK   0580
CNVINUM  055E      CNVSSNM  056C     CONCRLF  048B#     CREATE   0491#
DEND     01C8      DIG1     01C5     DIG2     01C6      DIG3     01C7
DIV16    0570#     DONE     04E3     ENDTK    05C4      ERROR    04F2
ERXIT    04EC      EXTMSG   0178     FCB01    04E4#     FNTFCB   047C#
GETDIG   054B      GETINUM  0526     GETSECT  05AD#     INDBUF   01E8
INDBUFX  0268      INDSW    01E7     INUM     01C3      MSG1     0192
MSG2     01B3      NOCARY   0599     NOINDP   04B8      NOTEND   05C3
NTINDRC  050F      ODDINM   04C8     OPEN     0000#     PRNTFN   0488#
PUTMC    04ED#     RD1SECT  0584     RD4SLP   0514      RDBLOCK  04F7
RDINDPB  05C8      RDNXBK   04CA     RDNXIBK  04DC      SCTBUF1  0516#
SCTPTR   01C9      SECT     01E6     SECTRAN  01CA      SELECT   0588#
SETBUFF  058C#     SIGNON   0153     STACK    0153      START    0468
TRACK    01E5      USEFHLF  04AE     WRTBUF1  0520#


** Cross Reference Table **

Symbol      Value  Defn  References

ADCTBIN   X 0000#  0029  0244
BEGIN       0100   0036
CLOSE     X 0000#  0028  0183
CNVBLK      0580   0292  0144 0215
CNVINUM     055E   0268  0126
```

(Continued on next page)

UNIX to CP/M (Listing continued, text begins on page 20)

Listing One

```
CNVSSNM      056C    0278  0294
CONCRLF   X  0000#   0029  0117
CREATE    X  0000#   0028  0122
DEND         01C8    0072
DIG1         01C5    0069  0237 0242
DIG2         01C6    0070  0239
DIG3         01C7    0071  0241
DIV16     X  0000#   0028  0271 0279
DONE         04E3    0182  0205
ENDTK        05C4    0337  0335
ERROR        04F2    0189  0248 0283 0315 0317 0325
ERXIT        04EC    0186  0190
EXTMSG       0178    0051  0185
FCB01     X  0000#   0028  0109 0115 0121 0182
FNTFCB    X  0000#   0028  0111
GETDIG       054B    0254  0236 0238 0240 0257 0259 0261 0263
GETINUM      0526    0235  0124
GETSECT   X  0000#   0030  0324
INDBUF       01E8    0092  0128 0135 0136 0146
INDBUFX      0268    0095  0169 0342 0344 0346 0348
INDSW        01E7    0089  0171 0210
INUM         01C3    0066  0133 0147 0249 0268
MSG1         0192    0056  0113
MSG2         01B3    0061  0189
NOCARY       0599    0312  0309
NOINDP       04B8    0146  0141
NOTEND       05C3    0336  0329
NTINDRC      050F    0215  0212
ODDINM       04C8    0157  0149
OPEN      X  0000#   0029
PRNTFN    X  0000#   0030  0116
PUTMC     X  0000#   0030  0104 0114 0186
RD1SECT      0584    0299  0131 0221 0343 0345 0347 0349
RD4SLP       0514    0218  0229
RDBLOCK      04F7    0198  0160 0175
RDINDPB      05C8    0342  0145
RDNXBK       04CA    0158  0163
RDNXIBK      04DC    0174  0177
SCTBUF1   X  0000#   0031  0220
SCTPTR       01C9    0075  0287 0305 0327 0337
SECT         01E6    0086  0313
SECTRAN      01CA    0078  0306
SELECT    X  0000#   0030  0107 0224 0301
SETBUFF   X  0000#   0029  0303
SIGNON       0153    0046  0103
STACK        0153    0043  0101
START        0468    0101  0036
TRACK        01E5    0084  0284 0321 0331 0333
USEFHLF      04AE    0139  0137
WRTBUF1   X  0000#   0031  0226
```

End Listing One

Listing Two

```
                    0001  ;
                    0002  ;-------------------------------------------------------
                    0003  ;       NLFILTER **NEW LINE FILTER PROGRAM**
                    0004  ;         NLFILTER converts UNIX Newline (0AH),
                    0005  ;         into CR (0DH) + LF (0AH) for CP/M
                    0006  ;         Command line = NLFILTER U:filename.ext
                    0007  ;         The Filtered File will be filename.NEW
                    0008  ;         This program uses the ZSUBLIB Libary
                    0009  ;         for use with ZMAC and ZLINK
                    0010  ;-------------------------------------------------------
                    0011  ;
                    0012  ;       --ZSUBLIB ROUTINES USED BY NLFILTER--
                    0013          EXT     FCB01,FCB02,FNTFCB,GETCHR,WRTCHR2
                    0014          EXT     OPEN,CREATE,CLOSE,PUTMC
                    0015  ;
    (0100)          0016          ORG     0100H           ;NORMAL PGM ORG
0100  C38501        0017          JP      START
                    0018  ;--RESERVE STACK SPACE--
0103  (0050)        0019          DEFS    80
                    0020  STACK:
                    0021  ;
0153  0D0A          0022  SIGNON  DEFB    0DH,0AH
0155  3E3E2055      0023          DEFB    '>> UNIX to CP/M '
      4E495820
      746F2043
      502F4D20
0165  4E657720      0024          DEFB    'New Line Filter '
      4C696E65
      20466696C
      74657220
0175  50726F67      0025          DEFB    'Program <<'
      72616D20
      3C3C
017F  0D0A24        0026          DEFB    0DH,0AH,'$'
                    0027  ;
                    0028  ;---FILE EXT FOR FILTERED FILE--
0182  4E4557        0029  NEW     DEFB    'NEW'
                    0030  ;
                    0031  ;--START OF NLFILTER PROGRAM CODE--
0185  315301        0032  START   LD      SP,STACK        ;SET STACK POINTER
                    0033  ;--SEND SINGON MESSAGE--
0188  115301        0034          LD      DE,SIGNON
018B  CD0000#       0035          CALL    PUTMC
                    0036  ;--USE FNTFCB TO FETCH FILE NAME FROM COMMAND
                    0037  ;   LINE AND  TO INSERT IT INTO FCB01, FCB02,
                    0038  ;   AND TO INITIALIZE FCB01 AND FCB02---
018E  110000#       0039          LD      DE,FCB01
0191  215C00        0040          LD      HL,05CH         ;COMMAND LINE
0194  CD0000#       0041          CALL    FNTFCB
0197  110000#       0042          LD      DE,FCB02
019A  215C00        0043          LD      HL,05CH
019D  CD0000#       0044          CALL    FNTFCB
                    0045  ;--OPEN THE INPUT FILE--
01A0  110000#       0046          LD      DE,FCB01
01A3  CD0000#       0047          CALL    OPEN
                    0048  ;--CHANGE EXT OF FCB02
01A6  010300        0049          LD      BC,03
01A9  218201        0050          LD      HL,NEW
01AC  110900#       0051          LD      DE,FCB02+9
```

(Continued on next page)

UNIX to CP/M (Listing continued, text begins on page 20)

Listing Two

```
01AF  EDB0          0052            LDIR
                    0053  ;--CREATE OUTPUT FILE--
01B1  110000#       0054            LD      DE,FCB02
01B4  CD0000#       0055            CALL    CREATE
                    0056  ;---USE GETCHR TO GET A CHARACTER FROM
                    0057  ;    THE INPUT ASCII FILE--
01B7  CD0000#       0058  CHRLOP    CALL    GETCHR
                    0059  ;--CHECK FOR EOF CHR--
01BA  FE1A          0060            CP      01AH            ;EOF CHAR
01BC  2810          0061            JR      Z,DONE          ;DONE IF EOF
                    0062  ;--CHECK FOR UNIX NEW LINE (LF)--
01BE  FE0A          0063            CP      0AH
01C0  2007          0064            JR      NZ,NTNWLN       ;NOT 0AH
01C2  3E0D          0065            LD      A,0DH           ;ADD CR
01C4  CD0000#       0066            CALL    WRTCHR2
01C7  3E0A          0067            LD      A,0AH           ;NOW LF
                    0068  ;--USE WRTCHR2 TO PUT CHARS INTO THE
                    0069  ;    OUTPUT FILE--
01C9  CD0000#       0070  NTNWLN    CALL    WRTCHR2
                    0071  ;---LOOP UNTIL END OF FILE--
01CC  18E9          0072            JR      CHRLOP
                    0073  ;---CLOSE OUTPUT FILE--
01CE  110000#       0074  DONE      LD      DE,FCB02
01D1  CD0000#       0075            CALL    CLOSE
01D4  C30000        0076            JP      00H             ;BACK TO CP/M
                    0077  ;---END OF PROGRAM--
01D7  (0000)        0078            END

Total 2nd Pass Errors  =          0
JPs that could be JRs  =          0
Z80/8080 Parity Usage  =          0


** Symbol  Table **

CHRLOP    01B7      CLOSE     01D2#     CREATE    01B5#     DONE      01CE
FCB01     01A1#     FCB02     01CF#     FNTFCB    019E#     GETCHR    01B8#
NEW       0182      NTNWLN    01C9      OPEN      01A4#     PUTMC     018C#
SIGNON    0153      STACK     0153      START     0185      WRTCHR2   01CA#

** Cross Reference Table **

Symbol      Value  Defn  References

CHRLOP      01B7   0058  0072
CLOSE     X 0000#  0014  0075
CREATE    X 0000#  0014  0055
DONE        01CE   0074  0061
FCB01     X 0000#  0013  0039 0046
FCB02     X 0000#  0013  0042 0051 0054 0074
FNTFCB    X 0000#  0013  0041 0044
GETCHR    X 0000#  0013  0058
NEW         0182   0029  0050
NTNWLN      01C9   0070  0064
OPEN      X 0000#  0014  0047
PUTMC     X 0000#  0014  0035
SIGNON      0153   0022  0034
STACK       0153   0020  0032
START       0185   0032  0017
WRTCHR2   X 0000#  0013  0066 0070
```

End Listing Two

A Small-C Help Facility

One nice feature of a "user friendly" operating system is the presence of a help facility. Many operating systems have this feature, including CP/M 3.0. Help programs are also available through the CP/M user's group. This article presents a different kind of help facility, one that is hierarchical in operation, detailing information about topics that have subtopics, recursively.

This program is modeled after the Digital Equipment Corporation VAX VMS 3.0 help program. As presented, it allows nesting of subtopics to any predefined depth (default is 24, which can be changed), and when the carriage return is pressed without a topic being entered, the stack pops back one level. If the stack pops out completely, the program returns control to the calling routine. At each new stack level, the list of available topics is presented (except for initial entry). If an unmatched topic is entered, the program apologizes for not having information on the subject and again lists the available topics at the current stack level.

Because the program is stack oriented, it allows duplication of topics so long as that duplication is not on a level equal to the current branch of the topic tree. For example, the main tree branch might have topics such as HELP, SORT, PRINT, and EDIT, while the next level down for the SORT branch might contain HELP, PATTERN, and FILES as subtopics. The term HELP is present in both levels, but the program does not confuse the two. On the other hand, a scheme containing HELP, PATTERN, PATTERN, and FILES on the subtopic branch for SORT is invalid. Also note that any branch in the hierarchy may echo the term that was used to create the branch; that is, HELP HELP HELP would be a valid set, just as SORT HELP is valid in the above example.

by John Staneff

John Staneff, Route 1, Box 801, Ellensburg, Washington 98926.

Permission is granted for personal, noncommercial use only. Any use for profit or other commercial gain without written permission of the author is prohibited.

Copyright © 1983 by John Staneff. All rights reserved.

The programs included in this article implement a systemwide help facility. The data file presented as an example (Table I, page 41) contains information about various utilities, including Ed Ream's screen-oriented editor, here renamed ED2.

General Description

The programming is in Small-C. It compiles under the unmodified version distributed by The Code Works. Because it uses only the simple constructs made available through that subset of C, it should be highly portable. (A note to those readers who might object to my less than purist use of the C language constructs and various function-naming schemes: This program works! For me, the major accomplishment was to implement a hierarchical stack pop/push algorithm in a language where the GOTO simply doesn't exist. I have found through trial and error that for the most part Small-C doesn't like recursion. Therefore, I managed to accomplish the same thing through the judicious use of the WHILE statements, as the coded routines can attest.)

The structure of the programs is such that the essence of the help facility can be incorporated into any application program. The file HLP.C (Listing One, page 46) is an include file calling in HLPDEF.C (Listing Two, page 46) and HLPPGM.C. HLPDEF.C contains the appropriate parameters for the application, such as stack size, stack depth, etc. HLPPGM.C (Listing Three, page 46) contains the actual program code. All global data references and function names in the program are preceded by the letters HLP. In this way I hope to avoid conflicts with other programs.

The compiled program requires approximately 7K bytes. This may seem a bit much, but it includes a considerable amount of stack handling code. In computing the application program's memory requirements, the size of the data stacks (as defined in HLPDEF.C) also must be considered. In the listings the stacks are set up for a systemwide facility and require 24K bytes.

The program requires that two defined variables be set up by the application program. They are NULL and CR (ASCII null and carriage return). Because these should already be in the application program, the help facility simply borrows them.

One difficulty in using Small-C is that you cannot nest #includes. The programmer therefore is faced with a minor decision in compiling the application. Either the HLP.C file can be eliminated and the application can call the two includes (HLPDEF.C and HLPPGM.C) directly, or the C80 compiler can be used to specify the application program and HLP.C at two subsequent "Input File" prompts.

Technical Description

The internal data structure portraying the hierarchy represents a multi-link descent tree. Figure 1 (page 45) shows a conceptual view of the tree. Each node (except for the root node) is a textual description of the current position within the structure. The links, one or more per node, are the descent mechanism: they are the keywords that provide the path required to change position within the tree.

In Figure 1, the root node has several named descent paths. These links are siblings to one another: they each point to a textual node describing the then-current location. Thus the node at SORT, one level down in the hierarchy, is a general description of the sort program. Each sibling also contains a pointer to the first descent path from the newly defined node. This link, in its turn, acts exactly like the initial link from the root node.

Therefore, every link provides a descent path to a newly defined node; a node terminates every descent path; and there is a potential recursion beginning at every node. For example, in Figure 1 the node at the termination of the descent path made up of the keywords SORT, PATTERN, and FIELD will describe how a field is defined to the sort program.

The combination of keywords must always be unique. Even though a given keyword may be used in any other position in the path, no two sibling descent paths may have an identical keyword. The level of descent is not important. Both SORT HELP and EDIT HELP are valid, since SORT and EDIT are separate nodes, which means the two HELPs are not siblings.

Following the descent tree is a hierarchical process. In principle, the tree can be descended from branch to branch at any node in the current descent path simply by switching to another sibling at any level, from 0 (root) to n (deepest climbed). Thus the path SORT PATTERN FIELD (yielding a description of how to

Table I
An example of a Help File

HELP
The system HELP facility is provided to make information available to the system user. It may be configured for the novice's use in working with CP/M, or a specific application system. For the more experienced user, it can be made to provide a set of reminders for those functions which are not 'on the tip of the tongue'.
/*

HELP USING
To use the HELP program, enter one of the topics listed as ready to be displayed. (You may re-list those topics by entering '?' to the prompt.) As information is provided about a topic, you may note a list of further information available, and a request for a subtopic. If you enter a 'return' by itself, you will back-out of the subtopic level, returning to the same level as the topic just displayed. The topic/subtopic scheme is hierarchical, with up to 24 levels of subtopic information available. If you respond 'return' to the highest-level prompt ('topic'), the program will terminate.
/*

HELP NEW
You may create new, or modified, help text, by modifying the file, 'HELP.HLP'. The syntax of the file is as follows:

line 1 — The proper syntax of the topic/subtopic name
lines 2-(n-1) — The text of the topic/subtopic
line n — Must be '/*'
/*

HELP NEW NAME
The proper syntax of the topic/subtopic name is:

1 to n tokens, may be capitalized, in a single line, and separated by at least a single space between each token.

Tokens 1 to (n-1) must be duplicates to another, earlier defined, topic/subtopic name. (If only one token, it must be unique.)
/*

HELP NEW FILE
All topic/subtopic definitions must be in the same file (HELP.HLP). Each set of lines 1-n (at least 3 lines per set), must follow each other in the file, one after the other.
/*

.
.
.

ED2
The full screen editor. This program can be used to modify any ASCII stream file (such as source programs). It provides a preview of the file being edited, each line shown in context with its neighbors in the file.

This program is a modified version of that as presented by Edward K. Ream in Dr. Dobbs Journal, Jan-82.

There are three basic modes of operation: Command, Edit, and Insert.
/*

ED2 COMMAND
Command mode is the most powerful editing mode, providing several functions which can affect the entire file at once.
/*

ED2 COMMAND APPEND
This command inserts a named file into the buffer at the current cursor position. Format:

append file.ext

ED2 COMMAND CHANGE
This command makes changes to lines within a specified line range (if not given, then the whole file). The changes are of a substitution nature, replacing each occurrence of a pattern string with another string. Format:

change <range>
search mask? mask
change mask? new
/*

.
.
.

ED2 COMMAND TABS
This command is used to set the current editing session's tab-size. The default is a tab every 8 characters. Use of this command can change that to any specified value. Note: use of non-standard tab settings can produce unpredictable results when a file is printed outside of the editor. Format:

tabs <n>
/*

ED2 EDIT
This mode allows manipulation within the text buffer, and correspondingly changes the CRT screen. There are several commands available for EDIT mode:
/*

ED2 EDIT SPACE
This command positions the cursor one position to the right, except if it is already at the extreme rightmost position of the current line. Format:

type a space
/*

ED2 EDIT B
This command positions the cursor to the beginning of the current line. Format:

type a 'b'
/*

ED2 EDIT C
This command places the editor into command mode. Format:

type a 'c', or alternatively, type a 'esc'

.
.
.

ED2 EDIT X
This command allows substitution of a specified character for the one at the cursor position. It is a one-character only substitution. Format:

type a 'xy' where y is the change mask
/*

ED2 INSERT
This mode allows for actual insertion of text. All special characters are operable in this mode, but the EDIT mode commands do not function. Text is collected until either the buffer is full, or the editor is returned to some other mode.
/*

ED2 SPECIAL
There is a collection of special edit commands which operate both in EDIT and INSERT mode. These perform functions which are not mode dependent, but rather are useful in any mode. Some of these commands are also allowed in COMMAND mode. Most of them leave you in EDIT mode, however a few will redirect you to one of the other modes:
/*

ED2 SPECIAL UP
This command allows you to reposition the cursor upward one line (unless it is already at the beginning of the text buffer). The editor is placed into EDIT mode. Format:

type a 'control-r'
/*

Table I
An example of a Help File

(Continued from page 41)

ED2 SPECIAL DOWN
This command allows you to reposition the cursor downward one line (unless it is already at the end of the text buffer). The editor is placed into EDIT mode. Format:

type a 'control-d'
/*

ED2 SPECIAL LEFT
This command allows you to reposition the cursor one character closer to the beginning of the current line (unless it is already at the beginning of the line). The editor remains in the current mode. This command may also be used in the command mode. Format:

type a 'left-arrow'
/*

ED2 SPECIAL RIGHT
This command allows you to reposition the cursor one character farther from the beginning of the current line (unless it is already at the extreme right of the CRT). The editor remains in the current mode. Format:

type a 'right-arrow'
/*

ED2 SPECIAL INSERT
These commands allow you to add lines in a prior existing text:
/*

ED2 SPECIAL INSERT UP
This command moves the edit buffer down one line and allows you to start entering text. The new text is AHEAD of the prior text where the cursor had been before the command. Note that the insert is by line, the editor will not break apart the current line before making room for the new text. The editor is placed into INSERT mode. Format:

type a 'up-arrow'
/*

ED2 SPECIAL INSERT DOWN
This command moves the cursor down one line and pushes the text after the current line down two lines, creating a blank line where you can insert new text. The new text is BEHIND that of the prior text where the cursor had been before the command. Note that the insert is by line, the editor will not break apart the current line before making room for the new text. The editor is placed into INSERT mode. Format:

type a 'return'

Note: during the normal course of entering text, pressing 'return' is actually enacting the special insert down command!
/*

ED2 SPECIAL DELETE
These commands allow deletions within the text body:
/*

ED2 SPECIAL DELETE LINE
This command deletes the line on which the cursor currently rests. There is no effect on edit mode. This command may be used in COMMAND mode. Format:

type a 'del'
/*

ED2 SPECIAL DELETE CHARACTER
This command deletes the character AHEAD of the cursor (unless the cursor is positioned at the beginning of a line). There is no effect on edit mode. This command may be used in COMMAND mode. Format:

type a 'back-space'

Note: This command is normally used while entering text in the INSERT MODE.
/*

ED2 SPECIAL UNDO
This command is useful for restoring a current line's original value when you have begun to change it, but decided to leave it as it was. There is no change to the edit mode. This command may be used in COMMAND mode. Format:

type a 'control-x'
/*

ED2 SPECIAL INSERT
This command places the editor into INSERT mode. The insertion begins AHEAD of the current cursor position. Note: a line will not wrap to the next if sufficient characters are entered to push it to the extreme right of the CRT. Therefore, when inserting into an existing line, the rightmost characters of the line may have to be deleted to make room for the new text. (One assumes they will be re-entered later. . .) This command may be used in COMMAND mode. Format:

type a 'control-n'
/*

ED2 SPECIAL EDIT
This command places the editor into EDIT mode. The cursor is positioned at the same place as it was last. This command may be used in COMMAND mode. Format:

type a 'control-e'
/*

ED2 SPECIAL COMMAND
This command places the editor into COMMAND mode (even if it's already there). The current location of the cursor is remembered. Format:

type a 'esc'
/*

ED2 SPECIAL SPLIT
This command splits a line into two pieces, the leftmost from the cursor remaining where it was, and the rightmost from the cursor moving to the next line downward. The editor is placed in INSERT mode. Format:

type a 'control-s'
/*

ED2 SPECIAL JOIN
This command joins two lines into one, that is the current line and the line immediately above it. Caution: if there isn't sufficient room to display the new joined lines on the CRT as a single line, the JOIN is not performed! Format:

type a 'control-p'
/*

End Table I

define a field to sort) can become SORT HELP (yielding a description of how to link at the SORT node. In practice, the program descending the tree must first "back off" from the deepest node in the path to one where the alternate descent path can be followed.

Operation

Two main function invocations are required of the application program, the first to hlpinit("file.dat") and the second to hlphelp( ). Both are illustrated in the HELP.C listing (Listing Four, page 69). The initial call sets up the internal tables by reading in the named sequential file. The help call may be issued any number of times, depending on the application logic. A decision was made to not display the available topics upon entry to the help call, presuming that the user could be expected to know on what subject help is required. Should the user type in a keyword not established as a help level topic (or type a question mark, provided no topic has been named with a question mark), the program will respond with a list of available subjects.

Normal operation for the user, once the request for help is made, is to con-

Table II
A sample run of the Help program
(User input is in italics)

```
B>
B>help
Help System Program

Topic? help
The system HELP facility is provided to make information available
to the system user. It may be configured for the novice's use in
working with CP/M, or a specific application system. For the more
experienced user, it can be made to provide a set of reminders for
those functions which are not 'on the tip of the tongue'.

Additional information is available for

USING
NEW
Sub-Topic? using
To use the HELP program, enter one of the topics listed as ready
to be displayed. (You may re-list those topics by entering '?' to the
prompt.) As information is provided about a topic, you may note a
list of further information available, and a request for a subtopic. If
you enter a 'return' by itself, you will back-out of the subtopic
level, returning to the same level as the topic just displayed. The
topic/subtopic scheme is hierarchical, with up to 24 levels of sub-
topic information available. If you respond 'return' to the highest-
level prompt ('topic'), the program will terminate.
Sub-Topic? new
You may create new, or modified, help text, by modifying the file,
'HELP.HLP'. The syntax of the file is as follows:

    Line 1 — The proper syntax of the topic/subtopic name
    lines 2-(n-1) — The text of the topic/subtopic
    line n — Must be '/*'

Additional information is available for

NAME
FILE
Sub-Topic? name
The proper syntax of the topic/subtopic name is:

    1 to n tokens, may be capitalized, in a single line, and separated
    by at least a single space between each token.

    Tokens 1 to (n-1) must be duplicates to another, earlier de-
    fined, topic/subtopic name. (If only one token, it must be
    unique.)
Sub-Topic? file
All topic/subtopic definitions must be in the same file (HELP.HLP).
Each set of lines 1-n (at least 3 lines per set), must follow each
other in the file, one after the other.
Sub-Topic? other
Sorry, no information is available for OTHER

Additional information is available for

NAME
FILE
Sub-Topic? <cr>

Additional information is available for

USING
NEW
Sub-Topic? <cr>

Additional information is available for

HELP
SORT
PIP
PRINT
ED2
Topic? ed2
The full screen editor. This program can be used to modify any
ASCII stream file (such as source programs). It provides a preview
of the file being edited, each line shown in context with its neigh-
bors in the file.

This program is a modified version of that as presented by Edward
K. Ream in Dr. Dobbs Journal, Jan-82.

There are three basic modes of operation: Command, Edit, and
Insert.

Additional information is available for

COMMAND
EDIT
INSERT
SPECIAL
Sub-Topic? command
Command mode is the most powerful editing mode, providing sev-
eral functions which can affect the entire file at once.

Additional information is available for

APPEND
CHANGE
CLEAR
DELETE
EXIT
FIND
G
LIST
LOAD
NAME
QUIT
SAVE
SEARCH
TABS
Sub-Topic? <cr>

Additional information is available for

COMMAND
EDIT
INSERT
SPECIAL
Sub-Topic? <cr>

Additional information is available for

HELP
SORT
PIP
PRINT
ED2
Topic? <cr>

B>
```

End Table II

tinue to enter topics (or subtopics) until the requested information has been delivered. When the user is finished, the program can be aborted (if stand-alone) by a control-C. If the help facility is incorporated into an application program, then pressing the carriage return without entering a topic will cause the descent tree logic to climb back one level. Should this retreat return the program to the root node, the application program once again will take control. (The sample help program run included in Table II, page 43, returns to CP/M at this point.)

The application programmer who includes the help facility must determine the size of its internal arrays, based upon the previously discussed considerations. Then an external help file must be prepared. The name of that file is transferred to the facility via the hlpinit call as the function's argument. The external file must not exceed the internal array storage.

How to Build a Help File

Help files are built using a text editor. These are ASCII text files with the normal CP/M line and file termination characteristics. They can be created using the stan-

dard ED editor. I have used Ed Ream's screen editor to create them without experiencing any difficulties.

Text to be considered by the program as a help file should conform to the following:

(1) The text that comprises the node is called a packet.

(2) Each packet consists of three parts: the header, the body, and the trailer.

(3) The whole help file is an assemblage of properly formatted packets.

A header is a single line of text (normally all upper case) composed of a string of words that uniquely identifies the packet. If there are n words in the line, then the first (n-1) words must have been previously defined, and the nth word must be unique in its position. (A word is defined by being the nth word in a header packet.) Definitions of the first (n-1) words must occur in a logical sequence prior to the definition of the nth word. For example, if a packet is to be defined for SORT PATTERN FIELD, then packets must already exist, in sequence, for SORT and for SORT PATTERN. In this example, SORT PATTERN must follow SORT. Other definitions may be interleaved, such as for SORT ORDER, so long as SORT precedes both SORT PATTERN and SORT ORDER, and SORT PATTERN precedes SORT PATTERN FIELD.

The body of the packet is composed of as many lines of text as are required to convey the node's meaning. This is displayed followed by a line for each subtopic keyword (descent tree link from the current node). Following the display of the subtopic keywords is a query asking for the user's further request. The size of the body may be limited by the intended display device. If a hard-copy terminal is used, there may be no limit to its size. When a CRT is used, however, the whole display (from original topic/subtopic request, through body and subtopics, to the new query) will probably need to fit in 24 lines.

The trailer is a single two-character string containing "/*". It is used as a separator between packets. Only this separator line can contain "/*" in its first two positions.

Summary

Since an operating system based in Small-C was presented in *DDJ* No. 77 March 1983, this help utility may be of significant interest to the *DDJ* community. In concert with the screen-oriented editor and current versions of Small-C, a fairly complete operating system environment can be constructed. Having used Small-C to build sort and print programs, I am extremely grateful to *DDJ* for the opportunity both to experiment with Small-C and to contribute to the growing body of published programs in Small-C. The multi-branch tree structure of the internal data representation for building and displaying help file texts theoretically could end up controlling a multi-level directory hierarchy in a Small-C based operating system or serving as the foundation of a hierarchical, Small-C based, Data Base Management system. The initial loading of the data file from the diskette, which is strictly a function of the speed of data transfer from storage media, may take a significant amount of time (depending on the file size). Once loaded, however, the threading and display algorithms are quite fast.

DDJ

(Listings begin on page 46)

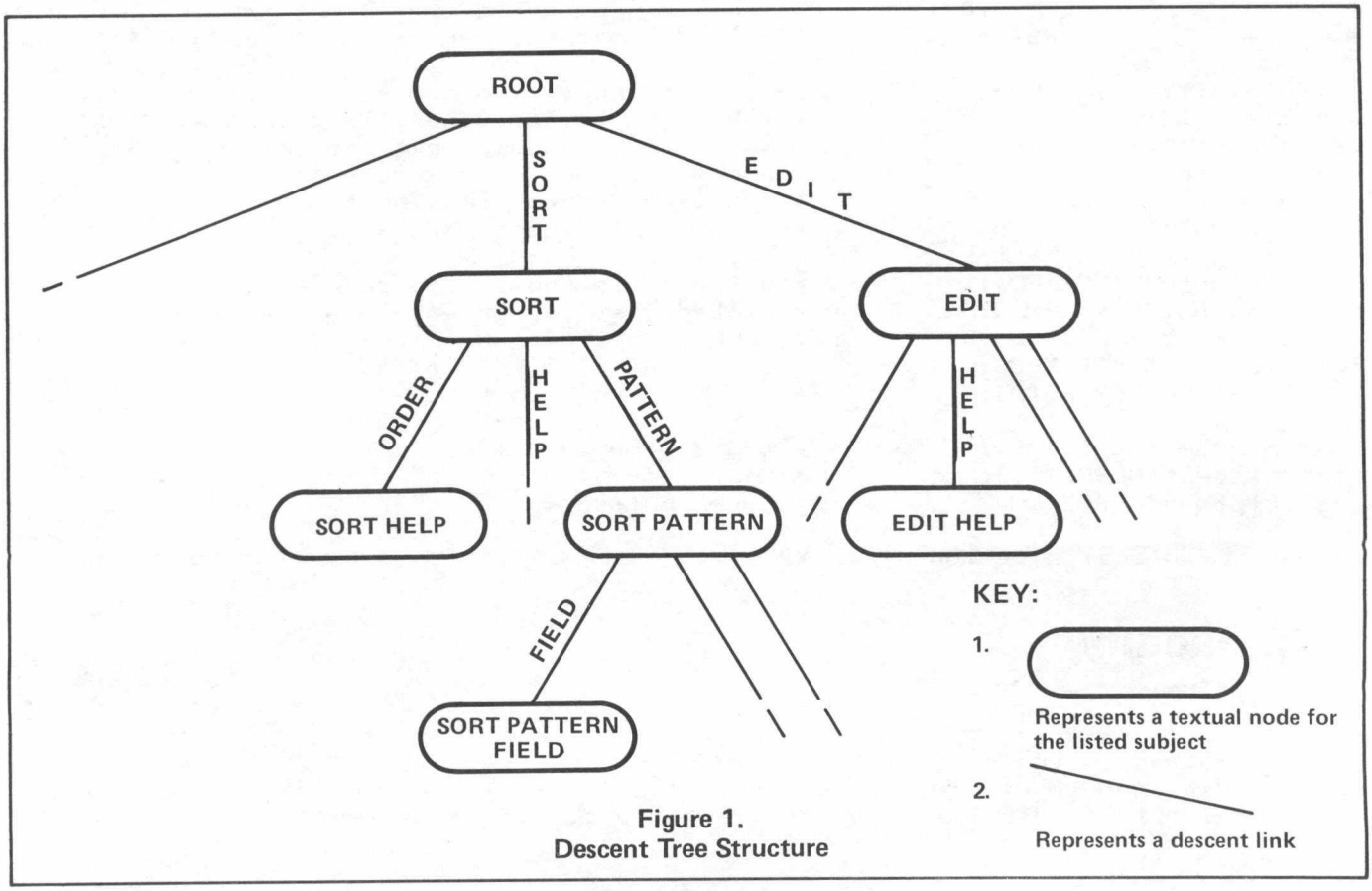


Figure 1.
Descent Tree Structure

Help Facility (Text begins on page 40)

Listing One

```
/*
 * Include file for HELP facility, if included in any program
 * then also brings in the subordinate files
 */

#include hlpdef.c        /* the particular custom tailored help sizes */
#include hlppgm.c        /* the programatic code of the facility */

/* end of the help include file */
```

End Listing One

Help Facility

Listing Two

```
/*
 * The custom tailored section of the HELP facility.
 * This include file should be evaluated for each new usage, and
 * modified to an appropriate size.
 * The buffers are sized for a standalone help program as
 * presented here.
 */

#define          HLPSZBUF          80       /* size of each text buffer */
#define          HLPSZTPC          200      /* # of topics available in memory */
#define          HLPSZNME          1000     /* # of bytes of topic name text */
#define          HLPSZTXT          24000    /* # of bytes of topic text */
#define          HLPNESTD          24       /* nesting level maximum */
#define          HLPSZTKN          160      /* token buffer size, 2*HLPSZBUF */

int hlpinfl;                                 /* input file number */
int hlptphsb;                               /* subscript to topic name text area */
int hlptptsb;                               /* subscript to topic text area */
int hlpnotpc;                               /* number of topics in memory */
int hlpnotkn;                               /* number of tokens in line */
int hlproot;                                /* root of all chains */

int hlptopic[HLPSZTPC];                     /* pointers to topic name text */
int hlpsbnxt[HLPSZTPC];                     /* linked list of current-level siblings */
int hlpsbtpc[HLPSZTPC];                     /* linked list of sub-topics */
int hlpptxt[HLPSZTPC];                      /* pointers to topic text */
int hlpptkn[HLPNESTD];                      /* pointers to each token */

char hlpname[HLPSZNME];                     /* topic name text */
char hlptext[HLPSZTXT];                     /* topic text */
char hlptokn[HLPSZTKN];                     /* token buffer */

/* end  of hlpdef.c include file */
```

End Listing Two

Help Facility

Listing Three

```
* This routine is the programatic code of the HELP facility.
* It is designed to display a hierarchical presentation of text
* from the file in hlpflnme, by topic/sub-topic.  It is called
```

```
 * at least twice in the calling program.  The first call to
 * hlpinit() reads in the data from the file, and establishes the
 * necessary internal pointers.  The second call to hlphelp() is
 * interactive with the user, responding to requests for topic/
 * sub-topic.
 */

hlpinit(hlpflnme)
char *hlpflnme;
{

int c;

/*
 * This is the appropriate call to cause the help facility to load
 * in the data file (hlpflnme) and build the internal pointer base.
 */

        c=hlpopen(hlpflnme);
        if (c!=0) {
                hlpbuild();
                fclose(hlpinfl);
        }
}

hlphelp()
{

/*
 * This is the appropriate call to query the user for topic/sub-topic
 * displays.  It returns when no further topics are entered.
 */

        hlpdispl();
}


hlpopen(hlpflnme)
char *hlpflnme;
{

/*
 * Open the help file containing the text which will be decoded
 * and stored.
 */

int c;

        hlpinfl=fopen(hlpflnme,"r");
        if (hlpinfl==NULL) {
                puts(hlpflnme);
                puts(" file not found");
                putchar(CR);
                return(0);
```

(Continued on page 50)

Help Facility (Listing continued, text begins on page 40)

Listing Three

```
        }
        return(1);
}

hlpbuild()
{

/*
 * Build topic table
 */

int switch;                                     /* 0=header, 1=text */
int l;
int c;
int p;                                          /* link-in switch 1=ok, else not */
char line[HLPSZBUF];                                    /* internal buffer */

        hlptphsb=1;
        hlptptsb=0;
        hlpnotpc=1;
        switch=0;
        hlproot=0;

        while ((l=hlpgtln(line))>0) {
                if (switch==0) {
                        hlpuprcs(line); /* be sure all uppercase */
                        /* first line of set is topic header */
                        hlpbltk(line);  /* extract all tokens */
                        switch=1;       /* flip switch */
                        p=hlplink();    /* link in this item */
                        if (p==1) {     /* record only if valid */
                                hlptopic[hlpnotpc]=hlptphsb;
                                hlpsvtpc(line);
                                hlpsbnxt[hlpnotpc]=0;
                                hlpsbtpc[hlpnotpc]=0;
                                hlpttxt[hlpnotpc]=hlptptsb;
                                hlpnotpc++;


                        }
                }
                else {
                        hlpsvtxt(line);
                        if ((line[0]=='/') & (line[1]=='*')) {
                                switch=0;       /* re-flip */
                        }
                }
        }
}

hlplink()
{

/*
 * Thread in this new topic/subtopic into the heirarchy
 */

int tst,c,r,thread,k;
int e,f;
char tkn[HLPSZBUF],tpc[HLPSZBUF];
```

```
int t,s,v;

        if (hlproot==0) {
                /* first time in */
                hlproot=hlptphsb;
                e=hlpptkn[0];
                return(1);
        }
        /* not first, find match if possible */
        thread=hlproot;
        k=0;
        while (k<=hlpnotkn) {
                r=0;
                tst=1;
                while (tst==1) {
                   /* get token name */
                   v=0;
                   t=hlpptkn[k];
                   while ((s=hlptokn[t++])!=NULL) {
                           tkn[v++]=s;
                   }
                   tkn[v]=NULL;     /* terminate it */
                   /* get topic at thread */
                   t=hlptopic[thread];
                   hlpgttpc(tpc,t);
                   /* compare token to topic */
                   c=hlpcmpr(tkn,tpc);
                   return(2);
           }

           /*
            * if here, cannot be  on the last token
            * and must be duplicate
            * until at bottom of hierarchy
            */
           if (r!=2) {
                   return(2);
           }
           /* still a duplicate from here on */
           else {
                   /*
                    * if no sub-topics and is next to last token,
                    * then link in as a new sub-topic
                    */
                   if (hlpsbtpc[thread]==0) {

                           /* nothing in heirarchy */
                           if (k==(hlpnotkn-1)) {
                                   hlpsbtpc[thread]=hlpnotpc;
                                   /* add a lower entry */
                                   return(1);
                           }
                           return(2);       /* quit too soon */
                   }
                   else {
```

(Continued on next page)

Help Facility

(Listing continued, text begins on page 40)

Listing Three

```
                        if (c==0) {
                                r=2;
                                break;
                        }
                        if (hlpsbnxt[thread]==0) {
                                tst=0;
                                r=1;
                        }
                        else {
                                thread=hlpsbnxt[thread];
                        }
                }

                /* if last token and not matched, link in */
                if ((k==hlpnotkn) & (r==1)) {
                        hlpsbnxt[thread]=hlpnotpc;
                        if (k==0) {
                                e=hlpptkn[k];
                                /* display sub-topics */
                        }
                        return(1);
                }
               /* if last token and still duplicate, report duplicate */
               if ((k==hlpnotkn) & (r==2)) {
                                /* start down the hierarchy */
                                thread=hlpsbtpc[thread];
                        }
                }
                /* now look at next token */
                k++;
        }

        /* if got here, MUST be a logic error */
        puts("Logic error????");
        putchar(CR);
        return(2);
}


hlpbltk(a)
char *a;
{

/*
 * Builds separate tokens, separated by spaces in source text
 */

int c,d,e,f,l;

        f=0;            /* 0=first char in token, else=1 */
        hlpnotkn=0;     /* number of tokens */
        d=0;            /* subscript to a[?] */
        e=0;            /* subscript to hlptokn[?] */
        l=0;            /* last character looked at */

        while ((c=a[d++])!=0) {
                if ((c==' ') & (l==' ')) {
                        l=c;
                }
```

```
                else {
                        if (f==0) {
                                f=1;
                                hlpptokn[hlpnotkn++]=e;
                        }
                        if (c==' ') {
                                hlptokn[e++]=NULL;
                                f=0;
                        }
                        else {
                                hlptokn[e++]=c;
                        }
                        l=c;
                }
        }
        hlptokn[e++]=NULL;
        hlpnotkn--;
}

hlpdispl()
{
/*
 * Display stored text for a match on topic keyword,
 * logic handles the hierarchical nature of these files.
 */

int ksub;
int k,c,tst,j,l;
char keywd[HLPSZBUF];                   /* input key word or phrase */
char line[HLPSZBUF];                            /* text buffer */

int retstck[HLPNESTD];                  /* return thread stack */
int stklvl,thread;
int txs;

        retstck[0]=hlproot;
        stklvl=0;

        tst=1;
        while (tst==1) {
                if (stklvl!=0) puts("Sub-");
                hlpgtprm("Topic? ",keywd);
                hlpuprcs(keywd);
                if (keywd[0]==NULL) {   /* if null, pop out */
                        if (stklvl==0) {
                                break;  /* done with this procedure */
                        }
                        stklvl--;
                        thread=retstck[stklvl];
                        hlpsubtp(thread);
                }
                else {
                        thread=retstck[stklvl++];
                        txs=1;
                        while (txs==1) {
                                ksub=hlptopic[thread];
                                hlpgttpc(line,ksub);
                                c=hlpcmpr(keywd,line);
                                if (c==0) {     /* display when equal */
```

(Continued on page 56)

Listing Three

```
                                        j=hlpttxt[thread];
                                        hlpdmptx(j);
                                        if (hlpsbtpc[thread]!=0) {
                                                thread=hlpsbtpc[thread];
                                                retstck[stklvl++]=thread;
                                                hlpsubtp(thread);
                                        }
                                        stklvl--;
                                        break;
                                }
                                thread=hlpsbnxt[thread];
                                if (thread==0) {
                                        puts("Sorry, no information is ");
                                        puts("available for ");
                                        puts(keywd);
                                        putchar(CR);
                                        thread=retstck[--stklvl];
                                        hlpsubtp(thread);
                                        break;
                                }
                        }
                }
        }
}

hlpsubtp(a)
int a;
{

/*
 * Print available subtopics at this level
 */

int thread,ksub;
char line[HLPSZBUF];

        putchar(CR);
        puts("Additional information is available for ");
        putchar(CR);
        putchar(CR);
        thread=a;
        while (thread!=0) {
                ksub=hlptopic[thread];
                hlpgttpc(line,ksub);
                puts(line);
                putchar(CR);
                thread=hlpsbnxt[thread];
        }
}



hlpsvtpc(a)
char *a;
{
/*
 * Save last token in name of topic
 */
```

```
int tst,c,d;
int v,t,s;
char tkn[HLPSZBUF];

        v=0;
        t=hlpttkn[hlpnotkn];    /* last token in name */
        while ((s=hlptokn[t++])!=NULL) {
                tkn[v++]=s;
        }
        tkn[v]=NULL;     /* terminate it */

        tst=1;
        c=0;
        while (tst==1) {
                d=hlptoupr(tkn[c++]);
                hlpname[hlptphsb++]=d;
                if (d==0) {
                        tst=0;
                }
        }
}

hlpsvtxt(a)
char *a;
{

/*
 * Save the text for a given topic
 */

int tst,c,d;

        tst=1;
        c=0;
        while (tst==1) {
                d=a[c++];
                hlptext[hlptptsb++]=d;
                if (d==0) tst=0;
        }
}

hlpgttpc(a,b)
char *a;
int b;
{



/*
 * Get topic name at location b
 */

int tst,c,d,e;

        tst=1;
        e=0;
        c=b;
```

(Continued on page 62)

Help Facility
Listing Three

(Listing continued, text begins on page 40)

```
        while (tst==1) {
                d=hlpname[c++];
                a[e++]=d;
                if (d==0) tst=0;
        }
}


hlpdmptx(a)
int a;
{
/*
 * Display the text for topic by location a
 */
int tst,c,b;

        b=1;
        tst=1;
        while (tst==1) {
                c=hlptext[a++];
                if (b==1) {
                        /* set on if prior char was null, or @ start */
                        if (c=='/') {
                                c=hlptext[a++];
                                if (c=='*') {
                                        tst=0;
                                        break;
                                }
                                else {
                                        puts("/");
                                }
                        }
                }

                b=0;
                if (c==NULL) {
                        b=1;
                        putchar(CR);
                }
                else {
                        putchar(c);
                }
        }
}


hlpgtln(lline)
char *lline;
{

/*
 * Gets input line, will return NULL if end of
   file, or
```

```
 * CR is at end of line
 */

int c,i;

        i=0;
        while ((c=getc(hlpinfl))>NULL) {
                lline[i]=c;
                if (c==CR) {
                        break;
                }
                else {
                        if (i<(HLPSZBUF-1)) {
                                i++;
                        }
                }
        }
        lline[i]=NULL;
        return(c);
}

hlpgtprm(m,r)
char *m,*r;
{


/*
 * Get input parameter from console
 */

        puts(m);
        gets(r);
}

hlpcmpr(a,b)
char *a,*b;
{
/*
 * Perform a FORTRAN-like compare.  Test a to b, and
 * return       -1 if a less than b,
 *               0 if a equals b, or
 *               1 if a greater than b.
 */

int r,s,t,q,c,d;

        r=0;
        q=1;
        s=0;
        t=0;
        while (q==1) {
                c=a[s++];
                d=b[t++];
```

(Continued on next page)

Help Facility

(Listing continued, text begins on page 40)

Listing Three

```
                if (c<d) {
                        /* first is low */
                        r=-1;
                        q=0;
                }
                if (c>d) {
                        /* first is  high */
                        r=1;
                        q=0;


                }
                if (c==0) {     /* stop if first over */
                        q=0;
                }
                if (d==0) {     /* stop if second over */
                        q=0;
                }
                /* Note, if c and d were equal, and are now zero,
                 * then logically the whole of each string has been
                 * found.......
                 */



        }
        return(r);
}

hlptoupr(c)
int c;
{


/*
 * convert a character to uppercase
 */

        if ((c<'a') | (c>'z')) {
                return(c);
        }
        else {
                return(c-32);
        }
}

hlpuprcs(a)
char *a;
{

/*
 * convert a string to upper case
 */

int c,d;

        c=0;
        while ((d=a[c])!=NULL) {
                a[c++]=hlptoupr(d);
```

```
        }
}


/* end of hlppgm.c include module */
/*
 * System HELP routine.
 *
 * This program is intended to be used by the novice
 * user to obtain information about the system by
 * typing HELP to the CP/M prompt.
 *
 * It queries the user for a topic (and subsequent
   subtopics)
 * of interest.  It is heirarchical in its
   approach to
 * response.
 *
 * The program is coded as a skeletal example of what
   is required
 * of an application program to incorporate the help
   facility.
 */

#define NULL    0       /* ascii for nothing */
#define CR      13      /* carriage return */

main()
{

        hlpinit("HELP.HLP");

        puts("Help System Program");
        putchar(CR);
        putchar(CR);

        hlphelp();

}

/*
 * Now programmer should include the hlp.c program in
   the c80 compiler
 * dialog.  Note the hlp.c contains includes,
   and c80 does not allow
 * nesting of includes.
 */

/* end of help utility */
```

End Listing Four

Attaching a Winchester Hard Disk To the S-100 Bus

by Oscar Goldman

It recently has become possible to purchase a multi-megabyte Winchester drive for a less-than-astronomical price. For instance, Priority One Electronics is selling a Micropolis 8-inch drive with a formatted capacity of 35 MBytes for slightly more than $4,000. This is complete with power supply, cabinet, and, most important, an intelligent controller.

The drive consists of three two-sided platters or six sides, one of which is reserved for supplying servo information for the voice coil head positioner. Each of the five data surfaces is divided into 580 tracks of 12K capacity. These tracks are further subdivided into sectors. When ordering the drive, you must specify the desired number of sectors; 24 sectors of 512 bytes or 12 of 1024 seem reasonable.

Because of the controller, interfacing the drive to the S-100 bus is very simple. A circuit for that purpose, suitable for wire-wrapping, will be described in this article. The controller also simplifies the necessary software for reading from and writing to the drive.

The Controller

Before discussing the interface, we need to know something about the controller. Even though I shall present a reasonably detailed description, it is essential that a potential user get a copy of Micropolis's documentation. (I purchased the Micro-Disk Maintenance Manual from them.)

The controller, which is Z80 based, handles all transactions between the computer and the drive. Connections between it and the host computer are made at a 34-pin edge connector on which all odd-numbered pins are grounded. The remaining 17 pins are arranged in sets as follows: eight bidirectional I/O data lines; two strobe lines for read and write (strobes are generated by the host); and one selector line (also from the host) that determines the nature of the information on the bidirectional bus, i.e., whether it is COMMAND/STATUS or READ/WRITE data. The remaining six lines are not of immediate importance for our purposes.

Because the data bus is bidirectional, you must pay careful attention to the timing of the read and write strobes. What is involved will become apparent when we discuss the interface board.

The Interface

The bidirectional bus goes to inverting buffers at the controller end and hence must be inverted at the interface. The 74LS240 tristate buffers (see Figure 1, page 73) are well-suited for use at the interface end of these lines; not only are they inverting, but they also supply (and sink) adequate current both for the interconnecting lines and for the S-100 bus.

The controller imposes certain timing restrictions on the read and write strobes. For writing (from host to drive), data must be placed on the I/O bus, and the write strobe must be active for at least 250 ns and then terminated before the data on the bus become inactive. To achieve these conditions, the write strobe is generated by two one-shots. The first, which has no timing capacitor, delays its output by a barely measurable interval before activating the second. The second one-shot has the duration of its pulse adjusted by a variable resistor to end just before WRITE/ goes high, which deactivates the data on the bus. The best way to adjust the timing of the second one-shot is to use a two-channel, triggered-sweep oscilloscope. One channel shows the level of WRITE/, the other channel the output of the second one-shot. In my setup, with a 4MHz Z80 CPU, the write strobe pulse can be adjusted to lie barely, but entirely, within the pulse representing the level at WRITE/. It is essential that you use a good quality timing capacitor; a disk ceramic would be a poor choice.

The situation for the read strobe is slightly different. The process involves activating the read strobe, reading the data, and terminating the strobe after the host has stopped reading. One way to achieve this is to note that, serendipitously, ADDSEL and SINP overlap PDBIN. The implementation of this scheme should be apparent from Figure 1.

The rest of the schematic is quite simple. The pin numbers of the ICs have not been marked; choose them to fit the arrangement of the chips on the interface board. Note that the pull-up resistors may range from 1K to 4.7K and are best supplied as a resistor pack. The designation of the lines at the controller end is taken from the Micropolis manual.

The interface board requires two adjacent ports. As shown in Figure 1, these ports have been wired as 7EH for COMMAND/STATUS and 7FH for READ/WRITE data. The actual port addresses may be changed by inserting additional inverters in the address lines. However, the even address must be kept for COMMAND and the odd one for data.

The Software

You can determine the state of the Winchester by reading STATUS on the COMMAND/STATUS port: whether it is busy, ready to transmit data, ready to receive data, and so on is determined by the state of the flags (see the Listing, page 74). In passing, I should note that the information contained in the Micropolis manual is not entirely clear and shows some inconsistencies between the text and the accompanying flow charts. A certain amount of experimentation was necessary before I arrived at the software included here. It works, but it may not be what Micropolis intended!

The Winchester is activated by the transmission of a command byte, followed by six parameter bytes, and then triggered by one more byte (whose actual value is unimportant). The manual mentions an optional capability that enables you to reread the bytes after they are transmitted to be sure that no transmission error has occurred. (You certainly do not want a read command to be misinterpreted as a write command!) I experimented with this and would suggest that "optional" be replaced by "essential." It is not clear where the problem arises, but some transmission errors do occur; rereading is a cheap form of insurance.

Complete software to run the Winchester drive would be difficult to present, since you would have to customize it for your own environment. The read/write module I have shown may be modified easily and incorporated into any 8080 or Z80 system. The listing illustrates some of the more attractive features of the controller: buffered read/write, automatic

Oscar Goldman, Department of Mathematics, University of Pennsylvania, Philadelphia, Pennsylvania 19104.

Copyright © 1983 by Oscar Goldman. All rights reserved.

Permission is granted for personal, noncommercial use only. Any use for profit or other commercial gain without written permission of the author is prohibited.

track seek before reading/writing, and reading/writing of multiple sectors. The controller is also capable of reading and writing "on the fly," but this requires a much more elaborate interface board.

No timing measurements have been made. However, I offer the following subjective impression: I am presently using double-density double-sided 8-inch floppies in Qume drives, formatted with 1K sectors and with separate read and write buffers, and the Winchester has about the same speed relation to my floppies as my floppies have to single-density single-sided ones.

Acknowledgement

I would like to take this opportunity to thank Pattie Kendall and Tim Matthews of *Micropolis* for their help in getting me past some early problems with the Micropolis Winchester drive.

DDJ

(Listing begins on page 74)

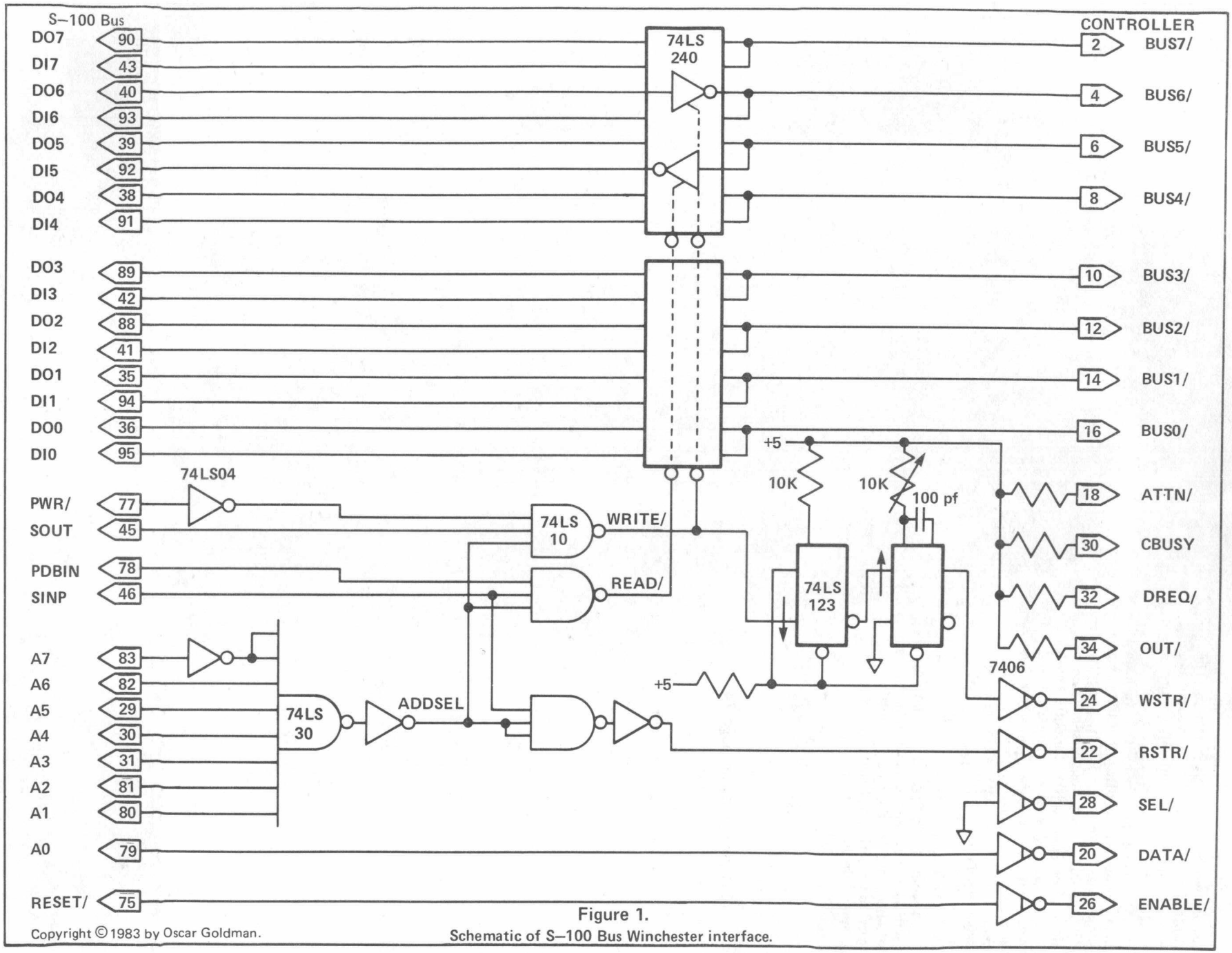


Figure 1.
Schematic of S–100 Bus Winchester interface.

Copyright © 1983 by Oscar Goldman.

Winchester Listing (Text begins on page 71)

```
        TITLE   'READ/WRITE MODULE'

;       Basic read and write routines for the Micropolis
;       Winchester drive and custom interface board.
;       Written March 1983 by Oscar Goldman


;       HARDWARE DEPENDENT DEFINITIONS

COMMAND EQU     7EH     ; Command/status port - output
STATUS  EQU     7EH     ; Command/status port - input
DATA    EQU     7FH     ; Bidirectional data port

;       STATUS MASKS

IRDY    EQU     1       ; 1 means a character is ready for the host
ORDY    EQU     2       ; 1 means the host may send data to the drive
HBUSY   EQU     16      ; Ø means that the drive is busy
DREQ    EQU     32      ; 1 means that drive is requesting data
ATTN    EQU     128     ; 1 means that the drive has completed its task

;       COMMAND CODES

RDCOM   EQU     4EH     ; Buffered read with auto seek and retry
WRCOM   EQU     47H     ; Same for write

;       MISCELLANEOUS DEFINITIONS

FALSE   EQU     Ø
TRUE    EQU     NOT FALSE

DEBUG   EQU     TRUE    ; Initial debugging is surely necessary

SECT$SIZE       EQU     512     ; May be changed to suit system

;*****************************************************************

HARD$READ:

;       Read one or more sectors into memory, starting at the
;       address stored in READ$ADD.
;       Assumes that the contents of READ$AREA have been prepared
;       before entry into the routine.

;       First send the read command information to COMM.

        MVI     C,RDCOM
        LXI     H,READ$AREA
        CALL    COMM

;       Now do the actual read.
```

(Continued on page 76)

Winchester Listing (Listing continued, text begins on page 71)

```
        LDA     RDSECTS         ; Number of sectors to read
        MOV     D,A             ; Save it
        LHLD    READ$ADD        ; Start storing it here
R1:     LXI     B,SECT$SIZE
R1Ø:    IN      STATUS
        ANI     DREQ+ATTN
        JZ      R1Ø             ; Wait for things to stabilize
        ANI     ATTN
        JNZ     DONE$READ       ; Done if ATTN says so
R11:
;       Read one sector
        IN      DATA            ; Get a byte
        MOV     M,A             ; Store it
        INX     H               ; Next location
        DCX     B
        MOV     A,B
        ORA     C               ; Finished?
        JNZ     R11             ; No

        DCR     D
        JNZ     R1              ; Next sector
DONE$READ:
;       Finished with the reading process; now
;       do the final cleanup.

        IF      DEBUG
        LXI     H,READ$AUX
        ENDIF                   ; Save exit information for
                                ; debugging purposes
        JMP     HARD$END

;*****************************************************************

HARD$WRITE:
;       See the description of the READ routine.

        MVI     C,WRCOM
        LXI     H,WRITE$AREA
        CALL    COMM

;       Now do the actual write.

        LDA     WRSECTS         ; Number of sectors to write
        MOV     D,A             ; Save it
        LHLD    WRITE$ADD       ; Start writing from here
W1:     LXI     B,SECT$SIZE
W1Ø:    IN      STATUS
        ANI     DREQ+ATTN
        JZ      W1Ø             ; Wait for things to stabilize
        ANI     ATTN
        JNZ     DONE$WRITE      ; Done if ATTN says so
W11:
```

```
;       Write one sector

        MOV     A,M             ; Get a byte
        OUT     DATA            ; Write it
        INX     H               ; Next location
        DCX     B
        MOV     A,B
        ORA     C               ; Finished?
        JNZ     W11             ; No

        DCR     D
        JNZ     W1              ; Next sector

DONE$WRITE:
;       Finished with the writing process; now
;       do the final cleanup.

        IF      DEBUG
        LXI     H,WRITE$AUX
        ENDIF                   ; Save exit information for
                                ; debugging purposes
        JMP     HARD$END

;*****************************************************************
COMM:
;       Command and parameter bytes are sent to the Winchester
;       and are immediately reread.
;       Enter with the command byte in the C register and the
;       data area pointed to by HL.


        PUSH    H               ; Save for emergencies
C0:     IN      STATUS
        ANI     HBUSY
        JZ      C0              ; Wait for not busy
        MOV     A,C             ; Get the command
        OUT     COMMAND         ; Send it
        CALL    HWAIT           ; Wait and reread
        CMP     C
        JNZ     C0              ; Bad reread - start over
        POP     H               ; Get it back
        PUSH    H               ; Save it once more
        MVI     B,6             ; 6 parameter byte
C1:     IN      STATUS
        ANI     ORDY
        JZ      C1              ; Wait for permission to send
        MOV     A,M
        OUT     DATA            ; Send out one parameter
        CALL    HWAIT           ; As before
```

(Continued on next page)

Winchester Listing (Listing continued, text begins on page 71)

```
        CMP     M
        JNZ     CØ              ; Bad reread - start anew
        INX     H               ; Next parameter
        DCR     B
        JNZ     C1              ; Continue
C2:     IN      STATUS
        ANI     ORDY
        JZ      C2
        OUT     DATA            ; This is the 'GO' byte
        POP     H               ; Don't mess with the stack
        RET                     ; All done


HWAIT:
;       Wait, reread the last byte sent
;       and return it in the A register.

HWØ:    IN      STATUS
        ANI     ORDY
        JNZ     HWØ             ; Wait for not ready
HW1:    IN      STATUS
        ANI     IRDY
        JZ      HW1             ; Wait for character to be available
        IN      DATA            ; Read it
        RET

;*************************************************************************
HARD$END:
;       All Winchester accesses end with this routine.
;       The 'termination' byte is returned in the E register,
;       and optionally, 13 auxiliary information bytes
;       useful for debugging, are stored in the data area
;       pointed to by HL.

EØ:     IN      STATUS
        ANI     HBUSY
        JZ      EØ              ; Wait for the dust to settle
E1:     IN      STATUS
        ANI     IRDY
        JZ      E1              ; Wait for a character to be ready
        IN      DATA            ; Get it
        MOV     E,A             ; and store it

        IF      DEBUG
        MVI     B,13            ; 13 aux bytes
E2:     IN      STATUS
        ANI     IRDY
        JZ      E2              ; Wait for a character
        IN      DATA            ; Get it
```

```
MOV     M,A             ; and store it
INX     H               ; Next one
DCR     B
JNZ     E2              ; Loop til done
ENDIF
RET

;*****************************************************************

;       DATA    AREA

READ$AREA
;       6 byte parameter area for read routine

RHEAD   DS      1               ; 16*head number
RTRACK  DS      2               ; 2 bytes for track
RSECT   DS      1               ; Starting sector number
RDSECTS DS      1               ; Number of sectors to read
        DB      Ø               ; Value doesn't matter for read

READ$ADD
        DS      2               ; Address to read to


WRITE$AREA
;       6 byte parameter area for write routine

WHEAD   DS      1               ; 16*head number
WTRACK  DS      2               ; 2 bytes for track
WSECT   DS      1               ; Starting sector number
WRSECTS DS      1               ; Number of sectors to write
        DB      Ø               ; Value doesn't matter for write

WRITE$ADD
        DS      2               ; Address to write from


        IF      DEBUG
READ$AUX
        DS      13
WRITE$AUX
        DS      13
        ENDIF
```

End Listing

Using The Epson Bit Plot Graphics

The Epson series of printers has been very popular, partly because of the variety of things they can do besides ordinary printing. One of the options for the MX-80 is the GRAFTRAX ROM, which gives the Epson a dot-level graphics capability. While this sounds like a powerful feature, it is not really all that easy to use, particularly since the MX-80 has no reverse line-feed command. Still it is possible, with a little patience, to get the printer to do some nice work.

This article describes how I used the Epson MX-80 to print electron density maps of the hydrogen atom. The calculations involved would keep most physics students busy for quite a while and would take those unfamiliar with the Schrodinger Wave Equation even longer. Therefore, this article does not set out to rigorously explain or solve the Schrodinger equation; many thorough treatments exist elsewhere and the particular forms of the equation I used may be found in the cited references. What I wish to present is the way in which I used the bit-plot graphics to represent the solutions.

The Problem

Most people are familiar with the basic structure of the atom: a central nucleus that is surrounded by "orbiting" electrons. The electrons are actually better described as a "cloud" because their exact positions can never be determined due to the Uncertainty Principle. What we can calculate, however, is the probability that an electron will be found in a particular region surrounding the atom. Many texts have "pictures" showing this electron cloud for the hydrogen atom (the only atom that is completely solvable) at different energy levels.

I wanted to be able to produce this type of electron density map. It seemed natural to break the problem into two parts: first calculating all the probability densities for a particular energy level and then using these values to produce the electron density map. The two programs *HYDROGEN* and HPLOT, which are written in Microsoft BASIC-80, achieve these two goals, respectively.

by Don Gay

Don Gay, Beulah College, P.O. Box 16, Nuku'alofa, Tonga.

The Program

The electron density map can be visualized as a grid of small probability cells with the nucleus at the origin (0,0). For each cell surrounding the nucleus a probability density is calculated. When the cell is printed its "darkness" is proportional to the probability at that point. Luckily, the plots are always symmetrical about the x and y axes so the cells need to be evaluated for only one quadrant; the other quadrants are obtained by simple reflection. Program HYDROGEN (Listing One, page 86) solves the Schrodinger equation and calculates the probability density at each (x,y) position. The program is written below in pseudo-code.

```
Input N,L,M;
Input scale factors;
Open output data file;
Output scale factors;
For each (x,y) coordinate {
    calculate probability density;
    convert to an ASCII character;
    write character to file;
}
Write highest probability obtained to file;
End.
```

The numbers N, L, and M are the three principal quantum numbers that determine which energy level the atom is in. These cannot be arbitrary but must conform to the following values: N = 1..MAX, L < N, M <= L. All three must be integers. The value of MAX in this program is 9, due to the way in which the output data filename is generated automatically from the values of N, L, and M.

Two scale factors are required. One is the maximum distance from the nucleus that you want to be calculated (RSCALE); this is in units of the Bohr radius, a useful unit in atomic measurements (~ 5.3E-10 meters). The other is the maximum probability density calculated in that plot. These factors are determined largely by trial and error. The program simplifies this task by printing at the end of a run the maximum probability calculated. Using this value as PSCALE, you can easily find the value of RSCALE so that the display starts at the edge of the specified grid. Table I (page 84) shows some values of RSCALE and PSCALE.

As the probability density for each cell in the positive quadrant is calculated, it is scaled to a relative value between 1 and 25 and converted to an ASCII character (A=1 and Y=25). These are printed out on the console device as they are calculated, as well as being written to the disk file, which makes it possible to see the progress of the calculation.

Figure 1 (page 84) shows a sample of the data file that HYDROGEN produces. Only the positive quadrant is present (compare this to the resulting bit plot in Figure 3 on page 85). The two values on the first line are RSCALE and PSCALE, respectively, and the number on the last line is the maximum probability calculated for that particular graph. The letter at the bottom left corner represents the origin, the left-hand column is the positive y axis, and the bottom row is the positive x axis of the grid.

Plotting the Data

The program HPLOT (Listing Two, page 92) takes as its input the data file created by HYDROGEN. It then prints out for each cell position on the entire grid a cell of dots whose darkness is proportional to the calculated probability density. At this stage we are dealing with only 25 different probability values, represented by the letters A - Y. Each cell comprises a matrix of dots with seven columns and eight rows; this is very nearly a square on the MX-80. Varying the number of dots printed in each cell for each probability density produces a cloud-like diagram. A dark cell contains many dots and corresponds to a high probability density, while a space or a light cell with only a few dots corresponds to a low probability density.

The Epson MX-80 printer has nine print needles in the print head, eight of which can be individually controlled. In the bit plot mode, you can instruct the printer to print up to 480 dots per line by sending an escape sequence that tells the printer how many of the next characters are to be interpreted as bit graphics data. As each character is sent to the printer, there is a direct mapping between the binary one's in the character and the print needles that fire.

For each cell that is printed, seven numbers, corresponding to the seven columns in the cell, must be sent to the printer. As each number is sent, one column of eight vertical dots is printed. The listing of HPLOT shows the specific codes required to print in the bit graphics mode. Note also that the line spacing is

changed so that adjacent lines touch each other to produce a smooth display.

I worked out the data needed for the 25 different cells on graph paper by filling in progressively more dots in each cell. The lightest cell is a space and each of the other cells has two more dots than the preceding one; the cell corresponding to the letter Y therefore has 48 dots. After plotting each cell on graph paper, I read the seven numbers required for each cell from the seven corresponding columns by designating a dot as one and a space as zero and reading the equivalent binary number. I found it quite difficult to produce a homogeneous looking cell for certain values, and some still have room for improvement. An example of this process is shown in Figure 2 (page 84). The x's in the graph image represent dots that are to be present. The periods indicate that no dot is to be printed. By reading down the vertical columns in the binary image you can see that the seven numbers required (in hex) are:

AA,5D,AA,77,AA,5D,AA

The data for the other letters can be seen in Listing Two. All these values are stored in an array, and the appropriate set of seven numbers to be sent to the printer is selected depending on the character (probability density) that is to be represented.

As HPLOT reads in the ASCII data file, it creates the full string image of the positive quadrant in memory. When plotting the other quadrants it simply takes the appropriate values that are stored in the positive quadrant. The size of the grid is from -30 to +30 cells in the x and y direction. The scaled distance, however, not the cell number, is printed on the graph axis.

I should also point out that, if the value of PSCALE you chose is smaller than the maximum probability calculated, the character @ will appear in the data file. This signifies an overflow and you should choose a larger value for PSCALE before attempting to plot the data. By varying RSCALE you can look at magnified views of the central portion if needed.

The number of calculations involved in both programs keeps the interpreter very busy indeed. In fact, when I first wrote the program using all single-precision variables for loop indexes, it took about 20 minutes to generate the data and as long again to print it out (2 MHz Z80). Since I have changed the variables to integers where possible, it now takes (depending on N,L,M) about 8-15 minutes to generate the data and about the same to print it out. I have also compiled the program using BASCOM, which enables both HYDROGEN and HPLOT to run approximately twice as fast. One thing the program seems to do quickly is use up ribbons on the printer!

Although the actual mathematics for obtaining the data are complex, the technique for representing the data with different density cells is quite general and could be applied to many different situations. One way of possibly speeding up HPLOT would be to convert all the input string data to an integer array that would contain the direct index into the dot graphics data array G%(). This would require less decoding than the present scheme, but I haven't bothered to make the change at this stage.

The references below give the formulae required for the solution of the Schrodinger equation for the hydrogen atom. Comments in the listing of HYDROGEN show where I obtained each formula.

References

1. H. L. Strauss. 1968. *Quantum Mechanics, An Introduction.* N.J.: Prentice Hall.
2. *Handbook of Tables For Mathematics.* 4th ed. CRC Press.
3. A. C. Wahl, P. E. Cade, and C. C. J. Roothan. November 1964. *J.C.P.*, vol. 41, no. 9, p. 2580.
4. *Handbook of Mathematical Functions.* 1972. ed. M. Abromowitz and I. Stegun. N.Y.: Dover.

(Listing begins on page 86)

Table I
Scale factors for different N, L, M combinations.

N,L,M	RSCALE	PSCALE	N,L,M	RSCALE	PSCALE
1,0,0	4.4	0.271	4,0,0	45	0.033
2,0,0	13.5	0.096	4,1,0	44	0.096
2,1,0	12	0.295	4,1,1!	44	0.048
2,1,1	12	0.147	4,2,0	42	0.163
3,0,0	27	0.051	4,2,1	36	0.061
3,1,0	26	0.153	4,2,2	42	0.061
3,1,1	26	0.078	4,3,0	37	0.245
3,2,0	23	0.268	4,3,1	35	0.087
3,2,1	20	0.100	5,2,0	63	0.118
3,2,2	23	0.101	5,2,1	56	0.043

```
20                .1
AAAAAAAAAAAABBBAAAAAAAAAAAAAAAA
AAAAAAAAABBBBBBBBBBAAAAAAAAAAA
AAAAAAABBBBBBBBBBBBBBAAAAAAAAA
AAAAABBBBBBBCCBBBBBBBBBBAAAAAAA
AAAABBBBCCCCCCCCCCBBBBBBBAAAAAA
AAAABBBCCCCDDDCCCCCCBBBBBBAAAAA
AAABBBCCDDDDDDDDDCCCCBBBBBBAAAA
AAABBCCDDEEEEEEEDDDCCCBBBBBBAAA
AAABCCDEEFFFFFFEEEDDCCCBBBBBAAA
AABBCDEFFGGGGGGFFEEDDCCCBBBBBAA
AABBCEFGHHIIIHHGGFFEDDCCCBBBBAA
AABCDEGHIJJJJJIIHGFFEDDCCBBBBBA
AABCEGHJKLLLLLKJIHGFEEDCCCBBBBA
AABDFHJKMNNNNNMLKJIHGFEDDCCBBBBA
AACDGIKMOPQQPONMKJIGFEEDCCBBBBA
AACEHJMOQRSSRQPNLKIHGFEDCCBBBBB
ABCFILOQSUUUTSQOMLJHGFEDDCCBBBB
ABCFJMPSUVWVUTRPNLJIGFEDDCCBBBB
ABDGJNRUWXXXVUSQNLJIGFEDDCBBBBA
ABDGKOSVXYYXWUSQNLJIGFEDCCBBBBA
ABDGKOSVXYYXVURPNLJHGFEDCCBBBBA
ABDGKORUWXXWUSQOMKIHFEDDCCBBBAA
ABDGJNQTUVVUSQOMKJHGFEDCCBBBBAA
ABCFILOQRSSRPOMKJHGFEDCCBBBBAAA
AACEGJLNOOONMLJIHGEEDCCBBBBAAAA
AABDEGIJKKKJJIHGFEDCCCBBBBAAAAA
AABBDEFGGGGGFFEDDCCBBBBBAAAAAAA
AAABBCCDDDDDCCCCBBBBBAAAAAAAAAA
AAAAAABBBBBBBBAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
.0997324
```

Figure 1.
Data file produced by the program HYDROGEN.

```
graph image                binary image

x . x . x . x              1 0 1 0 1 0 1
. x . x . x .              0 1 0 1 0 1 0
x . x x x . x              1 0 1 1 1 0 1
. x . x . x .              0 1 0 1 0 1 0
x x x . x x x              1 1 1 0 1 1 1
. x . x . x .              0 1 0 1 0 1 0
x . x x x . x              1 0 1 1 1 0 1
. x . x . x .              0 1 0 1 0 1 0
```

Figure 2.
The bit graphics data for the letter "Q".

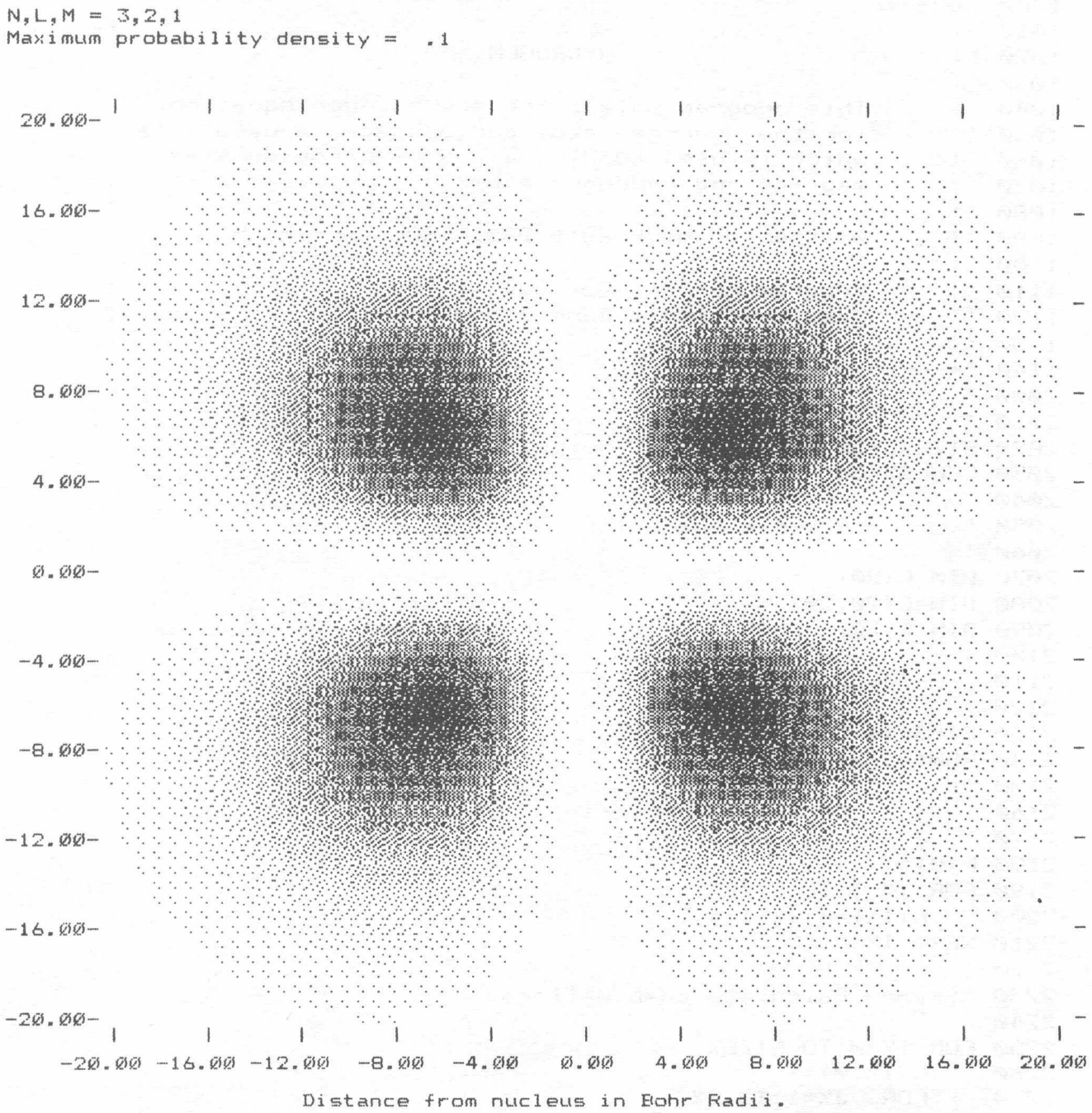


Figure 3.
Electron density map for the hydrogen atom.

Epson Graphics (Text begins on page 82)

Listing One

```
1000 '+-----------------------------------------------------+
1010 '¦                                                     ¦
1020 '¦                    --  HYDROGEN  --                 ¦
1030 '¦                                                     ¦
1040 '¦     This program solves the schrodinger equation    ¦
1050 '¦     for the Hydrogen atom and produces a data file  ¦
1060 '¦     which is used to plot a 2-D electron density    ¦
1070 '¦     map for the hydrogen atom.                      ¦
1080 '¦                                                     ¦
1090 '¦     Written in Microsoft BASIC-80  Version 5.2      ¦
1100 '¦                                                     ¦
1110 '¦     Program Author: Don Gay                         ¦
1120 '¦     Last Revision : June 7, 1983                    ¦
1130 '¦                                                     ¦
1140 '+-----------------------------------------------------+
2000 '
2010 '
2020 SIZE% =20                      'Size of factorial arrays.
2030 CELLS%=30                      'No. of cells in grid axis.
2040 '
2050 '------ Reserve space for variables.
2060 '
2070 DIM F(20)                      'Factorials
2080 DIM C(20,20)                   'Combinational Array
2090 DIM P(20)                      'Coefficients of Legendre
2100 '                                 .. polynomial
2110 '------ Define Constants.
2120 '
2130 PI=3.14159
2140 ROW$=STRING$(31,32)            'Initialize to 31 spaces
2150 '
2160 '------ Calculate Factorials.
2170 '
2180 F(0)=1
2190 FOR I%=1 TO SIZE%
2200    F(I%)=F(I%-1)*I%
2210 NEXT I%
2220 '
2230 '------ Calculate combinations.
2240 '
2250 FOR I%=0 TO SIZE%
2260    C(I%,0)=1
2270    FOR  J%=1 TO I%
2280            C(I%,J%)=F(I%)/(F(J%)*F(I%-J%))
2290    NEXT J%
2300 NEXT I%
2310 '
2320 '------ Get Quantum numbers & scale factors.
2330 '
2340 PRINT CHR$(27);"C"                     'Clear screen.
2350 PRINT TAB(24);"--  HYDROGEN  --":PRINT:PRINT
2360 INPUT "Enter values for N,L,M :",N%,L%,M%
2370 IF N%<=9 AND L%<N% AND M%<=L% GOTO 2375
2374 PRINT "Invalid entry, try again.":GOTO 2360
```

```
2375 'Form a 3 digit string from n,l,m for filename
2380 NLM$=MID$(STR$(N%*100+L%*10+M%),2,3)
2390 FILE$="HPLOT"+NLM$+".DAT"
2400 INPUT "Enter the radial scale factor :",RSCALE
2410 INPUT "Enter the probability scale factor :",PSCALE
2420 '
2430 '------ Evaluate constant term for this n,l,m choice
2440 C1=SQR((2*L%+1)*F(L%-M%)/(2*F(L%+M%)))
2450 '
2460 '------------ Store coefficients of Legendre polynomial
2470 '
2480 '  - first do special case Plm(0)
2485 '     (see Ref. 4, pg 334, eqn 8.6.1)
2490 GARG=(L%+M%+1)/2              'Gamma function argument
2500 GOSUB 4000:GAM1=GAMMA        'Calculate Gamma function
2510 GARG=(L%-M%)/2+1
2520 GOSUB 4000: GAM2=GAMMA
2530 P0=2^M%/SQR(PI)*COS((L%+M%)*PI/2)*GAM1/GAM2
2540 '  - now do all other coefficients of
2545 '     .. polynomial in cos(theta)
2550 '     (see Ref. 3, pg 2580, eqn II.19,II.20)
2560 FOR I%=0 TO INT((L%-M%)/2)
2570    P(L%-M%-2*I%)=(-1)^(M%+I%)*C1*F(2*(L%-I%))/
                (2^L%*F(I%)*F(L%-I%)*F(L%-M%-2*I%))

2580 NEXT I%
2590 '
2600 OPEN "O",#1,FILE$
2610 PRINT #1,RSCALE,PSCALE
2620 '
2630 MAX.P=0     'Maximum probability density for this n,l,m
2640 FOR Y%=CELLS% TO 0 STEP -1
2650    FOR X%=0 TO CELLS%
2660            XPOS=RSCALE*X%/CELLS%:YPOS=RSCALE*Y%/CELLS%
2670            GOSUB 3000       'Calculate prob. density
2675            'Remember maximum probability
2680            IF PROB>MAX.P THEN MAX.P=PROB
2685            'Scale probability
2690            P=CINT(PROB/PSCALE*24)+1:IF P>25 THEN P=0
2700            MID$(ROW$,X%+1,1)=CHR$(P+64)
2710            PRINT CHR$(P+64);        'Show on console
2720    NEXT X%
2730    PRINT "  -";Y%:PRINT #1,ROW$
2740 NEXT Y%
2750 PRINT #1,MAX.P
2760 CLOSE #1
2770 PRINT "Function complete for ";FILE$
2780 PRINT "Maximum probability density is";MAX.P
2790 END
2800 '
3000 '-----------Subroutine to calculate probability density
3010 GOTO 3150
3020 '
3030 ' Needs :
```

(Continued on page 90)

Listing One

```
3035 '   XPOS,YPOS = coordinates of required point
3040 '   N%,L%,M%   = principal quantum numbers
3050 '   F(n)       = n!
3060 '   C(n,r)     = n!/(r!(n-r)!)
3070 '   C1         = constant term for spherical harmonics
3080 '   PØ         = Legendre polynomial Plm(Ø) special case
3090 '   P(n)       = Coefficients of Legendre polynomial
3100 '   PI         = 3.14159
3110 '
3120 ' Returns :PROB       = Probability density
3130 '
3140 '--- Evaluate constant terms for this x,y
3150 IF XPOS<1E-5 THEN THETA=PI/2 ELSE THETA=ATN(YPOS/XPOS)
3155 'Convert to spherical coordinates.
3160 R=SQR(XPOS*XPOS+YPOS*YPOS)
3170 SINE=SIN(THETA) : COSINE=COS(THETA)
3180 LA=2*R/N%            'Argument for Laguerre polynomial
3190 '--- Calculate Laguerre polynomial (Ref. 2, pg 216)
3200 LGR=Ø
3210 FOR I%=Ø TO N%-L%-1
3220    LGR=LGR+(-1)^I%*C(N%+L%,N%-L%-I%-1)*LA^I%/F(I%)
3230 NEXT I%
3240 '--- Calculate Radial function (Ref. 1, pg 91)
3250 RAD=SQR(F(N%-L%-1)/(2*N%*F(N%+L%)))*
              (2/N%)^(L%+1.5)*R^L%*EXP(-R/N%)*LGR
3260 '--- Calculate Spherical Harmonics (Ref. 1, pg 75)
3270 'If cos(theta)=Ø or 1 then use special formula.
3275 '   (see Ref. 4, pg 334)
3280 IF COSINE=Ø THEN YSP=C1*PØ:GOTO 3340
3290 IF SINE=Ø AND M%<>Ø THEN YSP=Ø:GOTO 3340
3300 YSP=Ø
3310 FOR I%=Ø TO L%-M%:YSP=YSP+P(I%)*COSINE^I%:NEXT I%
3320 IF M%=Ø THEN 3340
3330 YSP=YSP*(-1)^M%*SINE^M%
3340 PROB=R*R*RAD*RAD*YSP*YSP
3350 RETURN
3360 '
4000 '------------Subroutine to calculate the Gamma function
4010 GOTO 4090
4020 '
4030 '   Needs:
4035 '     PI    = 3.14159
4040 '     F(n) = n!
4050 '     GARG = Argument must be integer or half-integer > Ø
4060 '
4070 '   Returns GAMMA = GAMMA(GARG)
4080 '
4090 IF GARG=INT(GARG) THEN Z=F(GARG-1):GOTO 4150
4100 GAMMA=1
4110 FOR I%=1 TO 2*(GARG-1) STEP 2
4120    GAMMA=GAMMA*I%
4130 NEXT I%
4140 GAMMA=GAMMA*SQR(2*PI)/(2^GARG)
4150 RETURN
```

End Listing One

Listing Two

```
1000 '+-------------------------------------------------------+
1010 '|                                                       |
1020 '|                    --  HPLOT  --                      |
1030 '|                                                       |
1040 '|   This program plots data generated by the program    |
1050 '|   HYDROGEN for the electron distribution around a     |
1060 '|   hydrogen nucleus for different N,L,M.               |
1070 '|                                                       |
1080 '|   Written in Microsoft BASIC-80 Version 5.2           |
1090 '|   Requires an EPSON MX-80 printer with GRAFTRAX.      |
1100 '|                                                       |
1110 '|   Program Author : Don Gay                            |
1120 '|   Last revision  : June 7, 1983                       |
1130 '|                                                       |
1140 '+-------------------------------------------------------+
2000 '
2020 '------ Reserve space for some variables.
2025 '
2030 DIM ROW$(30)                    'Ascii data for each row
2040 DIM G%(25,7)                    'Graphics data for each cell
2045 '
2050 '------ Define constants.
2055 '
2060 CELLS=30                        'No. of cells on each axis
2065 C1%=1:C64%=64
2070 '
2080 '------ Read bit graphics data
2085 '
2090 FOR I%=1 TO 25
2100     FOR J%=1 TO 7
2110             READ G%(I%,J%)
2120     NEXT J%
2130 NEXT I%
2140 '
2150 WIDTH LPRINT 255                'Inhibit auto CRLF on printer
2160 PRINT CHR$(27);"C"              'Clear screen
2170 PRINT TAB(22);"--  HPLOT  --":PRINT:PRINT
2180 PRINT "You may select to plot the electron ";
2185 PRINT "configuration for different"
2190 PRINT "values of n, l and m, the principal ";
2195 PRINT "quantum numbers.":PRINT
2200 PRINT "e.g. If you want a plot of n=2, l=1, m=0 then "
2210 PRINT "enter the string '210'.":PRINT
2220 INPUT "Enter the 3 numbers for n,l,m : ",NLM$
2230 IF LEN(NLM$)<>3 GOTO 2220
2240 FILE$="HPLOT"+NLM$+".DAT"
2250 OPEN "I",#1,FILE$                    'Get data file.
2260 INPUT #1,RSCALE,PSCALE
2270 FOR I%=CELLS TO 0 STEP -1
2280    INPUT #1,ROW$(I%)
2290 NEXT I%
2300 CLOSE #1:PRINT
2310 PRINT "Press <cr> when the printer is ready :";
2315 C$=INPUT$(1)
```

(Continued on page 94)

Epson Graphics (Listing continued, text begins on page 82)

Listing Two

```
2320 LPRINT TAB(17);
2325 LPRINT "Electron density map for the hydrogen atom."
2330 LPRINT TAB(17);STRING$(43,45):LPRINT:LPRINT
2340 LPRINT "N,L,M = ";MID$(NLM$,1,1);",";
2345 LPRINT MID$(NLM$,2,1);",";MID$(NLM$,3,1)
2350 LPRINT "Maximum probability density = ";PSCALE
2360 LPRINT:LPRINT
2370 LPRINT CHR$(27);"3";CHR$(24);'Set close line spacing.
2380 GOSUB 2670                    'Print x axis scale.
2390 FOR Y%=CELLS TO -CELLS STEP -1
2400    IF Y%/6<>INT(Y%/6) THEN LPRINT "       ";:GOTO 2420
2410    LPRINT USING "###.##";RSCALE*Y%/CELLS;:LPRINT "-";
2415    'Set printer to bit plot mode, 427 dots.
2420    LPRINT CHR$(27);"K";CHR$(171);CHR$(1);
2430    FOR X%=-CELLS TO CELLS
2440        P%=ASC(MID$(ROW$(ABS(Y%)),ABS(X%)+C1%,C1%))-C64%
2450        FOR C%=1 TO 7
2460            LPRINT CHR$(G%(P%,C%));
2470        NEXT C%
2480    NEXT X%
2490    IF Y%/6=INT(Y%/6) THEN LPRINT "-";
2500    LPRINT
2510 NEXT Y%
2520 GOSUB 2670                          'Print X axis scale.
2530 LPRINT:LPRINT "    ";
2540 FOR X=-CELLS TO CELLS STEP 6        ' .. and numbers
2550    LPRINT USING "###.## ";RSCALE*X/CELLS;
2560 NEXT X
2570 LPRINT CHR$(27);"@";                'Reset printer.
2580 LPRINT:LPRINT TAB(23);
2584 LPRINT "Distance from nucleus in Bohr Radii."
2585 LPRINT STRING$(12,10)               'Form feed
2590 PRINT "Would you like to print this out again";
2595 INPUT "(Y/N) ? ",A$
2600 IF A$="Y" GOTO 2310
2610 IF A$<>"N" GOTO 2590
2620 PRINT "Would you like to plot a different configuration";
2625 INPUT "(Y/N) ? ",A$
2630 IF A$="Y" GOTO 2160
2640 IF A$<>"N" GOTO 2620
2650 PRINT CHR$(27);"C"
2660 END
2670 '------------ Subroutine to print x axis scale.
2675 'Set printer to bit-plot mode, 462 dots.
2680 LPRINT " ";CHR$(27);"K";CHR$(206);CHR$(1);
2685 'Print vertical bars on axis
2690 FOR I%=1 TO 11
2700    LPRINT STRING$(41,0);CHR$(255);
2710 NEXT I%:LPRINT
2720 RETURN
3000 '-------------------------------------------------
3010 ' Bit graphics data for 480 dots per line
3020 '
3030 ' Each pixel is 7 columns X 8 rows (approx. square)
```

```
3040 '
3050 DATA &H00,&H00,&H00,&H00,&H00,&H00,&H00
3060 DATA &H00,&H00,&H20,&H00,&H04,&H00,&H00
3070 DATA &H02,&H00,&H20,&H00,&H04,&H00,&H40
3080 DATA &H40,&H02,&H08,&H00,&H10,&H40,&H02
3090 DATA &H04,&H40,&H10,&H02,&H80,&H08,&H21
3100 DATA &H24,&H80,&H12,&H40,&H04,&H10,&H41
3110 DATA &H42,&H08,&H22,&H84,&H11,&H40,&H12
3120 DATA &H88,&H22,&H48,&H25,&H80,&H12,&H48
3130 DATA &H51,&H04,&HA2,&H08,&H55,&H20,&H8A
3140 DATA &H92,&H24,&H49,&H92,&H24,&H49,&H92
3150 DATA &H88,&H55,&H02,&HA9,&H04,&HAA,&H55
3160 DATA &H42,&HAD,&H32,&HC8,&H12,&HB9,&H24
3170 DATA &HAA,&H41,&H36,&HC9,&H22,&H55,&HAA
3180 DATA &H55,&HA2,&H55,&HAA,&H55,&H8A,&H55
3190 DATA &H55,&HAA,&H55,&HAA,&H55,&HAA,&H55
3200 DATA &HAA,&H55,&HAA,&H77,&HAA,&H55,&HAA
3210 DATA &HAA,&H5D,&HAA,&H77,&HAA,&H5D,&HAA
3220 DATA &HDA,&H2D,&HD7,&HB7,&H6B,&HB4,&H5B
3230 DATA &HB6,&H6B,&HDA,&HB6,&H5B,&HED,&H56
3240 DATA &HDB,&H6D,&HB6,&HDB,&H6D,&HB6,&HDB
3250 DATA &H6A,&HDF,&HB5,&H6F,&HEA,&H9A,&HD5
3260 DATA &HEE,&HBB,&H77,&HBE,&HDD,&H77,&HDD
3270 DATA &H5D,&HFF,&HFA,&HDF,&H77,&HFE,&HAB
3280 DATA &HEF,&HDD,&H77,&HFF,&HDE,&H7B,&HEF
3290 DATA &HDF,&HFB,&H7F,&HED,&HFF,&HBE,&HF7
```

End Listing Two

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

DEC Professional 300 Microcomputers

I have some important (and expensively acquired) information for those of you looking speculatively at the new PC-300 series of personal microcomputers being sold by Digital Equipment Corp. Like everyone else who read the advertisements and saw the TV commercials, I was impressed by the styling and specifications of the machine, and found the idea of a PDP-11 compatible computer on a desktop running an operating system based on RSX-11M to be very appealing. As soon as the machine was officially announced, I placed my order for a PC-350 with the local DEC Product Center, and thereby hangs a sad tale.

When we placed an order in the summer, the salesperson estimated delivery to be in late September. After two months without a word, we called them back and were told it would arrive in late November. The new year came and went without a word from DEC, in spite of several letters I wrote trying to get some information about delivery status, and finally I gave up on the whole idea of buying a PC-350 and forgot about it.

Much to my surprise, in July 1983 (almost a year after the original order was placed) a truck with a DEC logo rolled up and dropped off three large boxes containing the keyboard, monitor, and system unit for a PC-350. Unpacking it, I began to get a little apprehensive when the chic grey two-inch binder and box titled "Professional 300 Diskette System" contained only a diagnostic diskette and a 50-page manual (consisting mostly of pictures) on how to hook up the various parts of the machine and plug it in. No sign of any other software, and the packing slip was marked "SHIP AS IS."

After assembling the PC-350 and verifying that it did indeed run the diagnostics, I began to investigate the little matter of the missing operating system. After all, the original salesperson claimed (and our purchase order clearly stated) *that a diskette-based operating system and development tools were to be included.* Suddenly the local DEC Product Center disclaimed all knowledge and responsibility for the PC-300 series, stating that marketing and support for those machines had been taken over by the main offices in Marlboro, Massachusetts. The original salesperson, as is usual for these computer horror stories, was no longer anywhere to be found.

Calls to other DEC dealers and OEM's in the area finally uncovered the amazing fact that (contrary to the original claims) no diskette-based operating system has been released at all! If you want a PC-350 that does anything, you have to buy the hard disk option, to the tune of another $5,000. The PC-325 is a dead issue, since it has no room for a hard disk expansion.

Even after you shell out additional money for a hard disk, you will not be able to do any program development on the PC-350. DEC apparently has no plans to make even an assembler and debugger available to run on the microcomputer itself. Instead, they want you to buy a "Programmer's Toolkit" that runs on a PDP-11/40 or VAX – the toolkit costs some $4,000, not counting the price of the VAX, of course. I guess the lesson to be derived from this story is that DEC is trying to make it impossible for the average small software house to produce packages for the PC-300 series, and I suspect that they are going to be quite successful in this objective. In the meantime, I have a very stylish grey DEC paperweight, and I don't think I'm going to pay for it.

Function Macros for the 8086/88

Ron Burk of Lawrence, Kansas, writes: "After following the discussion in the 16-Bit Software Toolbox about stack arrangements and the BP register, I came to the conclusion that this is a good thing to figure out once and then forget. Accordingly, I constructed the accompanying macros (Listing One, page 100), which mechanize some of the repetitiveness of 8088 programming. They are used to create, enter, and return from functions (or procedures if you prefer). The BP register is maintained in such a way that arguments to functions and local variables are located at fixed offsets from it.

"The package consists of five macros that are much easier to use than they were to make.

"The BENTRY macro defines the entry point of a function. You supply it with the name of the function and the names of the arguments to the function (which should have been pushed on the stack by the caller). It saves the current BP register on the stack and changes it to point to the top of the stack. It also defines the names of the arguments to the function as positive offsets to the BP, like [BP+2], [BP+4], etc.

"The AUTO macro defines 'automatic' storage for local variables to the function. These variables are offsets in the stack and automatically disappear when the function returns. The SP register is decremented in order to allocate the named variables, which are really negative offsets to the BP register (for example [BP-2]).

"The SAVE macro saves the named registers on the stack in a certain order. It also sets flags so that the BEND macro knows what (if any) registers have to be popped.

"The BEND macro defines the end of the function and pops off any saved registers in the correct order, then readjusts the stack. It also defines a label (consisting of the function name followed by _end) so that you can easily jump to the return code.

"Finally, the BCALL macro pushes arguments (if any) on the stack and then calls the named function.

"The macros include some rudimentary error checking in order to catch obvious mistakes in using them. They also print function names at the beginning and end of each function, which sometimes makes it easier to find mistakes. The following skeleton shows a prototypical use of the macros:

```
bentry    myfunc,<arg1,arg2,arg3>
auto      <local1,local2,local3>
save      <cx,dx,flags>
.
.

(body of function)
.
.

bend      myfunc
```

"In this example, the macros would cause the symbol arg1 to be equal to [BP+8]. The BEND macro expands into code to restore the saved registers and adjust the stack, then execute the proper RET instruction. It also defines the label myfunc_end to be used as a target for branches to the exit point from within the function.

"It is hard to construct a realistic example that is reasonably small, so I have enclosed a dummy example (Listing Two, page 106) that can be assembled by the readers to demonstrate the expansion of each macro. Although my assembler doesn't seem to mind, the assembler documentation states that it is an error to redefine a symbol via EQU. To avoid doing this, the macros employ some

trickery. If a local variable named FRED is to be defined, it is defined with an EQU to be [BP+FRED@], and FRED@ is defined (or redefined) to be the correct decimal offset via the '=' operator.

"Even for people who dislike one or more aspects of the macros, they are useful in strictly didactic sense, for they provide examples of many features of the MS-DOS assembler's macro facility. These macros are free to anyone who is willing to type them in."

More Low-Res Graphics for IBM PC

Robin Myers, of Livermore, California, writes: "In the December 1982 *DDJ*, you presented a method for obtaining low-resolution, 16-color graphics on the IBM PC. I have since tried the routine and found one very annoying problem: How do I get back to alphanumeric mode?

"Having the ability to display 16-color graphics is nice, but not very useful without a more general method of drawing in this mode. So I have included a set of BASIC subroutines (Listing Three, page 106), which support changing from 80 x 25 alpha text to 160 x 100 graphics mode and back again, point plotting, and line drawing. These routines were not optimized for execution speed, but were written to show the methods only."

Listing Four (page 111) is a simple demonstration program that draws a set of randomly corrected lines in the 16-color, low-resolution graphics mode.

An Appeal

Our thanks to Robin Myers and Ron Burk for a very educational set of contributions. Readers, please keep the programs coming! The lapse of time between your donation and its appearance in *DDJ* will be kept to a minimum if you send machine-readable source code on standard 8-inch CP/M or any IBM PC 5¼-inch disk format. Next month: 68000 Tools.

DDJ

(Listings begin on page 100)

16-Bit Toolbox (Text begins on page 98)

Listing One

```
;;
;; BMAC - Macros for 8088 Assembly Language
;; Ron Burk, June 1983
;;


@argdef  macro     argnames              ;; Internal mac for defining args
@nargs=0                                 ;; @nargs is # of bytes for arguments
ifnb<argnames>                           ;; If there were any arguments
    irp dummy,<argnames>                 ;; First, count # of args
 @nargs=@nargs+2                         ;; 2 bytes per argument
    endm
    @i=@nargs+2                          ;; Allow 2 bytes for return address
    irp arg,<argnames>                   ;; For each argument
 arg&&@=@i                               ;; Define a redefinable offset
 ifndef arg
   arg equ [bp + arg&&@]                 ;; Define name itself
 endif
 @i=@i-2                                 ;; Next arg is 2 bytes lower
    endm
endif
 endm                                    ;; End @argdef macro

@err macro msg                           ;; Print error message if in pass 1
if1
    %out msg
endif
 endm                                    ;; End of @err macro


@check macro extra,macname               ;; Checks for bad invocation
ifnb<extra>
   @err <You forgot angle brackets in macname macro>
endif
 endm                                    ;; End macro @check

bentry macro func,argnames,mistake       ;; Defines procedure
                                         ;; @nlv is # of bytes needed
@nlv=0                                   ;;  for local variables
@nsr=0                                   ;; # of bytes for saved regs
@savefl=0                                ;; Flag: save macro has been used
irp flag,<@ax,@bx,@cx,@dx,@si,@di,@ds,@ss,@es,@flags>
    flag=0                               ;; Initialize register flags
endm
ifb<func>
    @err <Missing function name on bentry macro>
else
    @err <!>!>!>Begin func>
endif
@check mistake,bentry
```

```
@argdef <argnames>                          ;; Define argument offsets
func proc                                   ;; Define a procedure
 push bp                                    ;; Save previous BP
 mov bp,sp                                  ;; Make new BP
 endm                                       ;; End bentry macro


auto macro vars,mistake                     ;; Defines local variables
if @savefl                                  ;; Must not use save before auto!
   @err <***ERROR*** must not save before auto>
endif
@check mistake,auto
irp var,<vars>
    var&&@=-(@nlv+2)                        ;; Add 2 to skip old BP
    ifndef var
      var equ [bp + var&&@]
    endif
    @nlv=@nlv+2
endm
 sub sp,@nlv                                ;; bump SP past local vars
 endm                                       ;; End auto macro
@pop macro reg                              ;; Internal macro to pop a register
if @&reg                                    ;; If we saved this reg
    ifidn<reg>,<flags>
 popf                                       ;; Use correct inst for flags
    else
 pop reg                                    ;; Or ordinary pop
    endif
endif
 endm                                       ;; End @pop macro
bend macro func                             ;; Define end of procedure
ifb<func>
    @err <Missing name on bend macro>
endif
@err <!<!<!<End func>
func&_end:                                  ;; Define an ending label
irp  reg,<flags,es,ss,ds,di,si,dx,cx,bx,ax>
    @pop reg
endm
if  @nlv                                    ;; If any local vars
    add sp,@nlv                             ;; Release local vars
endif
 pop bp                                     ;; Restore old BP
 ret @nargs                                 ;; Return and pop args
func endp                                   ;; Define end of procedure
 endm
```

(Continued on next page)

16-Bit Toolbox (Listing continued, text begins on page 98)

Listing One

```
@save macro r1,r2                       ;; Famulus of save macro
ifidn<r1>,<r2>                          ;; If they are the same
    @&r1=1                              ;; Set flag variable
    @found=1                            ;; Tell caller we found it
endif
 endm                                   ;; End @save macro
save macro regs,mistake                 ;; Macro for saving registers
@savefl=1                               ;; For error checking
@check mistake,save
irp  reg,<regs>
    @found=0                            ;; Haven't found reg yet
    irp  regname,<ax,bx,cx,dx,si,di,ds,ss,es,flags>
 @save  regname,reg                     ;; Set corresponding macro variable
    endm
    ife  @found                         ;; If we don't know that name
 @err <***ERROR*** 'reg': Unknown register name in save>
    endif
endm
irp  reg,<ax,bx,cx,dx,si,di,ds,ss,es,flags>
    if @&&reg                           ;; If we're supposed to save that one
 ifidn<flags>,<reg>
     pushf
 else
     push  reg
 endif
    endif
endm
 endm                                   ;; End save macro


bcall macro func,args,mistake           ;; Macro for calling procedure
@check mistake,func
ifnb<args>                              ;; If we're supposed to push anything
    irp arg,<args>
 push arg                               ;; Push whatever on the stack
    endm
endif
 call func
 endm                                   ;; End of bcall macro
```

Listing Two

```
        page    ,132
;
        .xlist
include         a:bmacs.asm
        .list
;
pgmseg segment
        assume  cs:pgmseg,ss:pgmseg,ds:pgmseg,es:pgmseg
main_entry:
        mov     bp,sp           ; Initialize BP
        bentry  main            ; Define main function
        auto    <len>
        nop                     ; Body of function 'main'
        cld                     ; goes here
        cld                     ; (whatever)
        bcall   fun1,<len>      ; Might call something else
        nop
        bend    main
;;;;
        bentry  fun1,<len>      ; Ok to use 'len' again
        auto    <a,b,c>         ; Local variables
        save    <flags,cx,es,bx>        ; Save registers if you want
        nop
        nop
        cld                     ; Body of function 'fun1'
        je      fun1_end        ; Label of return of 'fun1'
        nop
        bend    fun1


pgmseg ends
        end     main_entry
```

End Listing Two

16-Bit Toolbox

Listing Three

```
50000 REM entry point for point plotting
50010 GOTO 50120
50020 REM entry point for line drawing
50030 GOTO 50410
50040 REM entry point to set up low res graphics display
50050 GOTO 50970
50060 REM entry point to restore normal 80 x 25 alpha display
50070 GOTO 51160
50080 REM entry point to define misc functions
50090 GOTO 51260
50100 ' ********************************
50110 '
50120 ' DRAW A POINT
50130 '
50140 ' on entry set ZX,ZY to x,y coordinates for the point
50150 '               DOTCOLOR to point color
50160 '
50170 ' get the value currently at the byte location
50180 ' for the dot
50190 DOTNOW = PEEK ( FNOFFSET%( ZX,ZY ) )
50200 '
50210 ' if the new dot is to go into the left half of
50220 ' the byte, then mask off the upper nybble (preserving
50230 ' the lower nybble) then OR in the new dot
50240 '
50250 IF FNMOD2%( ZX ) <> 0 GOTO 50330
50260 DOTCLR = (DOTNOW AND &HF) OR (DOTCOLOR * 16)
50270 GOTO 50370
50280 '
50290 ' if the new dot is to go into the right half of the
50300 ' byte, then mask off the lower nybble (preserving the
50310 ' upper nybble) and OR in the new dot
```

(Continued on next page)

Listing Three

```
50320 '
50330 DOTCLR = (DOTNOW AND &HFØ) OR DOTCOLOR
50340 '
50350 ' display the dot (write it back to refresh buffer)
50360 '
50370 POKE FNOFFSET%( ZX,ZY ),DOTCLR
50380 RETURN
50390 '
50400 '
50410 ' DRAW A LINE
50420 '
50430 ' on entry set:  X1,Y1 to x,y coordinates for starting point
50440 '                X2,Y2 to x,y coordinates for ending point
50450 '                DOTCOLOR to color for entire line
50460 '
50470 ' compute the change in x and y
50480 '
50490 DELTAX = X2 - X1 : DELTAY = Y2 - Y1
50500 '
50510 ' Draw a point
50520 '
50530 IF DELTAX <> Ø OR DELTAY <> Ø GOTO 5Ø62Ø
50540 '
50550 ' Just draw a point if the line has zero length
50560 '
50570 ZX = X1 : ZY = Y1 : GOSUB 5ØØØØ
50580 RETURN
50590 '
50600 ' Check for case of horizontal line
50610 '
50620 IF DELTAY <> Ø GOTO 5Ø71Ø
50630 ZY = Y1
50640 FOR ZX = X1 TO X2 STEP SGN( DELTAX )
50650   GOSUB 5ØØØØ
50660   NEXT ZX
50670 RETURN
50680 '
50690 ' Check for case of vertical line
50700 '
50710 IF DELTAX <> Ø GOTO 5Ø8ØØ
50720 ZX = X1
50730 FOR ZY = Y1 TO Y2 STEP SGN( DELTAY )
50740       GOSUB 5ØØØØ
50750 NEXT ZY
50760 RETURN
50770 '
50780 ' Case for slope > 1
50790 '
50800 IF ABS( DELTAY ) < ABS( DELTAX ) GOTO   5Ø9ØØ
```

```
50810 SLOPE = DELTAX / DELTAY
50820 FOR ZY = Y1 TO Y2 STEP SGN( DELTAY )
50830 ZX = SLOPE * ( ZY - Y1 ) + X1
50840  GOSUB 50000
50850 NEXT ZY
50860 RETURN
50870 '
50880 ' Case for slope <=1
50890 '
50900 SLOPE = DELTAY / DELTAX
50910 FOR ZX = X1 TO X2 STEP SGN (DELTAX )
50920   ZY = SLOPE * ( ZX - X1 ) + Y1
50925  GOSUB 50000
50930 NEXT ZX
50940 RETURN
50950 '
50960 '
50970 ' Set up low resolution 160 x 100 16-color mode
50980 '
51000 SCREEN 0 : WIDTH 80                    ' set up 80 x 25 alpha
51010 KEY OFF : CLS                          ' clear screen
51020 OUT &H3D8,9                            ' disable blink attribute
51030 A% = &H3D4 : D%=&H3D5                  ' crt controller ports
51040 OUT A%,4 : OUT D%,&H7F                 ' vertical total
51050 OUT A%,6 : OUT D%,&H64                 ' vertical displayed
51060 OUT A%,7 : OUT D%,&H70                 ' vertical sync position
51070 OUT A%,9 : OUT D%,1                    ' max scan line address
51080 DEF SEG = &HB800                       ' display buffer segment
51090 FOR CC% = 0 TO &H3FFE STEP 2           ' all character codes preset
51100   POKE CC%,&HDE : NEXT                 ' to special character
51110 FOR AT% = 1 TO &H3FFF STEP 2           ' all attributes preset
51120   POKE AT%,0 : NEXT                    ' to black
51130 RETURN
51140 '
51150 '
51160 ' reset 160 x 100 graphics mode to 80 x 25 alpha mode
51170 '
51180 OUT A%,4 : OUT D%,&H1F                 ' vertical total
51190 OUT A%,6 : OUT D%,&H19                 ' vertical displayed
51200 OUT A%,7 : OUT D%,&H1C                 ' vertical sync position
51210 OUT A%,9 : OUT D%,7                    ' max scan line address
51220 CLS
51230 RETURN
51240 '
51250 '
51260 ' define miscellaneous functions for 160 x 100 graphics
51270 '
51280 ' compute byte offset into display buffer based on X,Y
```

(Continued on next page)

16-Bit Toolbox (Listing continued, text begins on page 98)

Listing Three

```
51290 DEF FNOFFSET%( X%,Y% ) = 160 * Y% + ( X% OR 1 )
51300 '
51310 ' modulo 2 function
51320 DEF FNMOD2%( X% ) = X% - ( INT( X% / 2 ) * 2 )
51330 RETURN
```

End Listing Three

16-Bit Toolbox

Listing Four

```
100 ' Sample program to produce set of randomly colored
110 ' lines, using 160 X 100 low resolution 16-color
120 ' graphics mode
130 '
140 GOSUB 50080                          ' define misc functions
150 GOSUB 50040                          ' set up 160 X 100 display
160 X1 = 0 : Y1 = 0 : X2 = 159
170 FOR Y2 = 0 TO 99 STEP 2
175   DOTCOLOR = INT(RND*15) : IF DOTCOLOR=0 OR DOTCOLOR=8 GOTO 185
180   GOSUB 50020
185 NEXT Y2
190 FOR X2 = 158 TO 0 STEP -2
195   DOTCOLOR = INT(RND*15) : IF DOTCOLOR=0 OR DOTCOLOR=8 GOTO 205
200   GOSUB 50020
205 NEXT X2
240 IF INKEY$ = "" GOTO 240              ' wait for any key
250 GOSUB 50060                          ' restore normal alpha mode
260 STOP
```

End Listing Four

CP/M EXCHANGE

by Robert Blum

When I talk about a program in this column that I am impressed with I usually try to summarize its features by including a chart of functions or at least a command summary. I adopted this method of review after years of requesting information on products and receiving sales brochures that gloss over any merits that the product may have. This type of brochure is fine for the uninitiated computer user, but does very little for me. What I want to see is a brief synopsis of what the product will do and how it might fit into my operating environment. Unfortunately, getting this type of information usually requires purchasing a demo system whose price may or may not be refundable if the complete system is bought later. The alternative to the demo system is to call the manufacturer and talk with a technical support representative – that is, if you can get through the switchboard without a serial number.

Well, that's the state-of-the-art! I do hope that the charts and such that I present here are helpful.

Hidden Treasures

A complete set of utility subroutines is absolutely necessary when writing anything more complex than a very simple assembler program – that is, unless you enjoy recoding all the fairly small, but necessary, CP/M interface and format conversion routines for each new program. Two solutions exist for this problem, and both are available for next to nothing in the public domain.

Several macro libraries have been in the public domain for quite some time that offer most of what you need. To use them requires a macro assembler such as MAC or M80. When using them, expect compile times to be longer than you are accustomed to, especially when defining several sequential files in the same program with the FILE macro. I have experienced delays of several minutes while the macros are being expanded. During this time, no file activity is taking place so you will have no indication as to whether the assembler has gone south or not. Be patient, it will finish in due time. When producing hard copy of the program, be prepared for a much larger listing than you may have expected. Now would be a good time to brush up on your assembler's switches or flag settings to suppress the statements generated through macro expansion. Setting one of these will trim down your listing to a more reasonable size. One drawback to using macros is that they may generate labels that can duplicate others that you have written. After a few uses, you will begin to remember what the generated labels are and avoid using them.

Rather than using macros, I prefer pre-assembled subroutines, which can be linked together with my program. Ron Conn has submitted to the SIG/M library the most complete set of subroutines that I have found. They are available on SIG/M volumes 88, 89, and 90, both in relocatable form (.REL files) and in source code form (.MAC files). Included with each routine is a documentation file that briefly, but completely, describes each routine's purpose and how to effectively use it. To use the relocatable version of them, you will need a linkage editor, such as LINK or L80, capable of accepting standard Microsoft relocatable files. This also means that you will have to compile your program with an assembler that produces this same type of object file.

If you decide not to use the relocatable version of SYSLIB, you can include the source code into your program. Again, when using this method, you run the risk of duplicating labels, and the assembly time will balloon. Whichever you choose, you can't go wrong with SYSLIB. For an abbreviated description of an older version of SYSLIB's contents, consult Table 1 (page 114).

TUTORI/O

Learning all the ins and outs of how CP/M works from the Digital Research documentation can be a painful experience. Many fine books have been written to help fill in the gaps, but nothing can rival actually watching the CPU's reaction to each instruction. I briefly talked about this subject several months ago and obviously feel just as strongly about it today.

Roy Lipscomb of Logic Associates wrote to me in response to that column. He pointed out that he had recently completed TUTORI/O – a program that would supposedly ease the task of learning how CP/M performs disk I/O operations. The documentation that he enclosed was intriguing. It explained that all the different disk-related BDOS functions could be exercised through single-letter commands, and the resulting status codes and save areas are automatically displayed for visual inspection. I was caught by my thoughts on how many hours I had spent pawing through piles of source code and sitting frustrated in front of a CRT attempting to relate CP/M's documentation to my program's operation. I wanted to see if TUTORI/O could live up to its billing; so I requested a copy for review.

Preparing TUTORI/O for use is simple. It is a single phase program; so only one program file and the documentation file had to be copied onto my backup diskette. I found out later that it requires the use of a debug program such as DDT, SID, or whatever. So, while creating your backup diskette, also copy whatever debugger you use as well. I first printed the documentation and gave it the once over. It is brief but complete, although it does assume that you have some prior knowledge of CP/M's BDOS functions. I found one area to be confusing at first; but after some further study, it became clear.

Executing TUTORI/O is as simple as typing its name followed by two optional parameters. The first is the name of your debug program followed by the line length of your CRT. If either parameter is omitted, a suitable default is automatically selected. The debug program is used to allow you to look into and change areas of memory other than those addressed by TUTORI/O.

Taking TUTORI/O to task, I loaded it specifying that I wanted to use SID as my debug program and that my CRT has an 80-column display. I was greeted with a menu of commands and a terse description of each. I first created a test file with the (M)ake command and was automatically presented with the return status and a hexadecimal display of the default FCB at 5CH and the default DMA area at 80H. Next, I sequentially wrote several sectors into the file. As each sector was written, I was informed of the BDOS return code and an updated display of the FCB. To put TUTORI/O to the acid test, I exited to my debug program, a single-letter command, and changed the random record number to a value well beyond what the first extent could handle. Returning back to TUTORI/O, again a single-letter command, I requested that the next record be randomly written into the file. A little more disk activity occurred this time while the first extent was written back to disk and a new one created. The resultant display was breathtaking. The record number that I chose had not only exceeded the first extent's capacity, but the second one as well. What I saw dis-

Table I
A Very Abbreviated List of SYSLIB Routines

Directory Manipulation Routines

DIRECT — loads a copy of the directory of the currently logged-in disk drive into the 32- x 64-byte buffer pointed to by HL selects all file entries which match the contents of the 11-byte file name buffer pointed to by DE (buffer may contain ?'s as wild specifications), and alphabetizes the entries in the buffer either by file name/type or by file type/name. Only entries where the extent is zero (1st file entry) are returned.

DIR — loads a copy of the directory of the currently logged-in disk drive into the 32- x 64-byte buffer pointed to by HL. On return, A contains a count of the number of entries in the directory.

Numeric Evaluation Routines

EVAL — converts the character string pointed to by HL into the 16-bit binary number represented by it. EVAL performs the conversion until a non-hexadecimal character is encountered, at which time it looks at the last character and the previous character to determine if the string is representing a binary, octal, decimal, or hexadecimal number. Input string characters may be upper or lower case. Valid formats for the input string are

bbbbbbbbbbbbbbbbB — b=0 or b=1; Binary string
ttttt or tttttD — 0 <= t <= 9; Decimal string
hhhhH or hhhhX — 0 <= h <= F; Hexadecimal string
oooooooO or ooooooooQ — 0 <= o <= 7; Octal string

Byte-Oriented File Input/Output

F0$OPEN, F10$OPEN, F20$OPEN, F30$OPEN — open the file whose FCB is pointed to by DE for output (use with F$PUT).

F0$CLOSE, F10$CLOSE, F20$CLOSE, F30$CLOSE — close the file previously opened by the corresponding F0$OPEN routine. Use of these routines is *mandatory* after output to the file (using the corresponding F$PUT routine) is complete.

F$GET, F1$GET, F2$GET, F3$GET — get the next byte in sequence from the file previously opened by the corresponding FI$OPEN routine. The byte is returned in Register A.

F$PUT, F1$PUT, F2$PUT, F3$PUT — put the byte in Register A onto the end of the file previously opened by the corresponding F0$OPEN routine.

CP/M File Input/Output Routines

F$OPEN — opens the file specified by the FCB pointed to by DE. If a file is not found, F$OPEN asks the user (on CON:) if he or she wishes to create it and does so if the answer is 'Y' or 'y' (F$OPEN simply returns with an appropriate error code if the answer is negative).

F$CLOSE — closes the file whose FCB is pointed to by DE.

F$MAKE — creates (Makes) the file whose FCB is pointed to by DE.

F$DELETE — deletes the file whose FCB is pointed to by DE. If a file does not exist, nothing happens (no error message or code is given).

F$READ — reads next block (128 bytes) from the opened file whose FCB is pointed to by DE into TBUFF (buffer at 80H-0FFH).

F$WRITE — writes next block (128 bytes) from TBUFF (buffer at 80H-0FFH) to the opened file whose FCB is pointed to by DE.

Input Line Editor

INLINE — allows the user to input a line of text from CON: into the buffer pointed to by HL. The user is allowed to edit the text as he or she types it.

String Output Routines

PRINT, LPRINT — print string pointed to by the return address on CON: (PRINT) or LST: (LPRINT). The string is terminated by a binary 0. Control is returned to byte following string.

PSTR, LPSTR — print string pointed to by HL on CON: (PSTR) or LST: (LPSTR). The string is terminated by a binary 0.

Numeric Output Routines

PHL4HC, LHL4HC — print HL as four (4) hexadecimal characters on CON: (PHL4HC) or LST: (LHL4HC).

MHL4HC — stores HL as four (4) hexadecimal characters in the 4-byte memory buffer pointed to by DE. On return, DE points to the byte following this buffer.

PHL5DC, LHL5DC — print HL as five (5) decimal characters on CON: (PHL5DC) or LST: (LHL5DC).

MHL5DC — stores HL at five (5) decimal characters in memory in the 5-byte buffer pointed to by DE. On return, DE points to the byte after this buffer.

PHLDC, LHLDC — print HL as up to five (5) decimal characters with leading spaces (<SP>) on CON: (PHLDC) or LST: (LHLDC).

MHLDC — stores HL as up to five (5) decimal characters with leading spaces in the 5-byte memory buffer pointed to by DE.

PA2HC, LA2HC — print A as two (2) hexadecimal characters on CON: (PA2HC) or LST: (LA2HC).

MA2HC — stores A as two (2) hexadecimal characters in the 2-byte memory buffer pointed to by HL. On return, HL points to the byte after this buffer.

PA3DC, LA3DC — print A as three (3) decimal characters on CON: (PA3DC) or LST: (LA3DC).

MA3DC — stores A as three (3) decimal characters in the 3-byte memory buffer pointed to by HL. On return, HL points to the byte after the buffer.

PADC, LADC — print A as up to three (3) decimal characters with leading spaces (<SP>) on CON: (PADC) or LST: (LADC).

MADC — stores A as up to three (3) decimal characters with leading spaces in the 3-byte memory buffer pointed to by HL. On return, HL points to the byte after this buffer.

String Comparison Routines

SCANNER — scans the vector of bytes pointed to by HL for the vector of bytes pointed to by DE. The HL-byte vector is B bytes long and the DE-byte vector is C bytes long. On return, if found, HL points to the beginning location within the original HL vector of the located vector and the zero flag is set; if it is not found, the zero flag is not set and HL is not affected (it points to the beginning of the original HL-byte vector).

General Input Routines

CONDIN — inputs a character from CON: if one is available; otherwise, returns a flag stating that a character is not available on CON:.

CST — inputs the status of CON: in Register A. If read data are available, A = 0; otherwise, A = 1.

CIN, RIN — input character from CON: (CIN) or RDR: (RIN) in Register A.

General Output Routines

COUT, LOUT, POUT — output character in Register A to CON: (COUT), LST: (LOUT), or PUN: (POUT).

CRLF, LCRF — print <CR> and <LF> on CON: (CRLF) or LST: (LCRLF).

Miscellaneous Routines

BDOS — calls CP/M entry point at location 5 and preserves Registers HL and DE.

CAPS — capitalizes an ASCII character in Register A if it is a lower case alphabetic character (a-z); otherwise, returns A unaffected.

CIO (CP/M I/O through BIOS Jump Table) — indexes into the CP/M BIOS jump table and allows the user to address any routine in the BIOS (accessible routines only). It is called with Register A containing the index offset and Registers BC containing any required input arguments.

EN — exchanges nibbles in Register A; *high-order* four bits are exchanged with low-order four bits of Register A.

FILLB, FILLBC — fill an area of memory with a constant byte value. FILLB can fill up to a 256-byte buffer, and FILLBC can fill up to a 65,536-byte (within reason) buffer.

MOVEB, MOVEBC — move the block of memory pointed to by HL to the memory location pointed to by DE. MOVEB can move up to a 256-byte buffer, and MOVEBC can move up to a 65,536-byte buffer.

CATH — converts the ASCII hexadecimal character in the A Register to binary in the A Register.

played was the third extent, and the block allocation number was sitting in the second position of the FCB. I hate to think about how long I spent learning how CP/M handled random record operations when version 2.2 first came out. Unfortunately, at that time I didn't have TUTORI/O and had to use some very crude methods to learn what I experienced in just several minutes with this program. TUTORI/O is a fine product aimed at the person who wants to learn more about the BDOS disk functions with a minimal amount of effort. To get further information or to place an order ($25.00 plus $6.00 handling), contact Roy Lipscomb, Logic Associates, 1433 W. Thome, Chicago, Illinois 60660.

Auto Disk Format Selection

Most of today's small computer systems accept at least two disk formats, single- and double-density. A few others allow as many as six. For example, the Tarbell double-density disk controller can successfully utilize 8-inch disks formatted in the standard single-density format of 26 128-byte sectors, and, in their own double-density format, of either 51 128-byte sectors or 16 512-byte sectors. The reason for using these nonstandard formats is to maximize data capacity while minimizing memory usage for data buffer space. In the case of the 16 512-byte sector format, each track carries the same amount of data as the standard 8 1024-byte sector format while saving 512 bytes of memory.

To enable the application program to access data from any one of the various formats requires that the BIOS be smart enough either to independently determine how a disk is formatted or to be instructed by the operator at the time a disk is mounted.

The Tarbell double-density disk controller uses the most straightforward approach to providing completely automatic disk type sensing. The disk initialization utility always formats the first track in single-density and then the remaining tracks in double-density. A single byte, contained in the bootstrap record placed on the disk at SYSGEN time, is then used to indicate format type. When a warm start occurs, the first sector of track zero, bootstrap record, is read to determine what the disk format is on all tracks other than zero. The CCP and BDOS are read from the second through 26th sectors of the first track in single-density and the remainder from track one in whatever format was indicated in the bootstrap record. In the case of logging in any drive other than A:, the disk selection routine again reads the bootstrap record to determine disk format and sets the appropriate indicators for subsequent use by the BIOS disk handling routines. One other automatic selection method that I have seen used is what I call the "brute force method." This method simply begins by reading the disk in single-density. If a read error occurs, the BIOS logic shifts to double-density and performs the read operation over agian. This tends to be a bit slow when logging in a double-density disk because a read error will always occur on the first read. This usually takes an additional 5 to 10 seconds depending on how laborious the error recovery routines are over normal log-in times.

The last, and most unsophisticated, method that I have seen requires that the operator identify the disk format and indicate to the system what it is through an operator command. For example, I recently had the opportunity to work on an Altos ACS8000, which uses this method of disk format selection. On this system you must use the MODE transient program to indicate to the system what drive is being changed and the disk format being mounted. Even so, this procedure should be used only when mounting a disk that is formatted differently than the one currently logged in to a particular drive.

Now that we have looked at the operational aspects of handling different disk formats, let's explore the BIOS logic necessary to interface between CP/M, the hardware, and the application program. The ability of a system to handle different disk formats comes from each attached physical drive being assigned a disk parameter header (DPH), sector translation table (STT), and a disk parameter block (DPB). It is through these tables that CP/M's BDOS can gain knowledge of how large the data space is, how many directory entries have been assigned, and other information necessary for it to successfully perform its job. The one key element that ties all the parts together is the disk parameter header. In it are contained pointers to all the static tables and work areas that are used when accessing a drive. Each drive attached to the system must have a DPH assigned to its exclusive use.

When logging in a disk through automatic selection, the first task of the BIOS is to determine what type of disk has been inserted by one of the two methods previously discussed. It is then necessary to update the DPH for this drive with pointers to the proper DPB and STT. Once this is complete, other indicators are also set to identify what the physical sector size is and what the hardware characteristics are. Since the DPB and STT are not modified during normal processing, only one for each different disk format is necessary. When two or more drives contain the same type of disk, their DPH will point to the same DPB and STT. In the auto-select BIOS of our example, a different DPB and STT for each disk type must be memory-resident at all times. This does impose some additional memory overhead because each DPB requires 15 bytes and each STT can be as large as 64 bytes.

When not using an auto-select scheme, the transient program called when mounting a new disk usually contains the tables

Table II
Character Devices

Label	Description	Contents
PDEVNM	Physical device name	Six characters describing the physical
MODE	Mode of operation	1 – device may do input 2 – device may do output 3 – device may do both input and output 4 – baud rates are software selectable 8 – device may use protocol 16 – XON/XOFF protocol enabled
BAUD	Baud rate	0 – none 1 – 50 baud 2 – 75 baud 3 – 110 baud 4 – 134.5 baud 5 – 150 baud 6 – 300 baud 7 – 600 baud 8 – 1200 baud 9 – 1800 baud 10 – 2400 baud 11 – 3600 baud 12 – 4800 baud 13 – 7200 baud 14 – 9600 baud 15 – 19.2K baud

Circle no. 103 on reader service card.

for each different disk type. Gaining access to the DPH – so that it can be changed to point to the appropriate tables – is accomplished by calling the BIOS entry point SELDSK. When a return to the transient program is made, the HL register pair will contain the address of the DPH for the selected drive. The transient program then moves the corresponding tables into the BIOS and updates any other work areas. This method does save a little memory, but requires operator intervention – not only to insert the new disk into the drive, but to log it in as well.

If you have the need to access the DPB from an application program, BDOS function 31 should be used. The DPB address of the current default drive is returned from this function call in the HL register pair. One other function call that may be of interest if you need system information on this level is BDOS function 27. The address of the allocation table for the current default drive is returned from this call in the HL register pair.

Additional information on the exact content of the tables that we have talked about can be found in the March and April issues of *DDJ* in this column.

CP/M Plus Device Tables

One design concept used when implementing CP/M Plus that was immediately useful to me is the extensive use of tables to describe the I/O system – in particular, the character device table that replaces the I/O byte of prior releases. Through this table, the characteristics of a large number of character-oriented devices can be initially described and later changed by a utility program to meet changing needs. This simplifies the controlling logic for each device because there is now a common format that can be used to determine exactly what driver software should be used.

In addition to simplified driver logic, it is now possible to easily assign one logical device to several physical devices by using a utility program. For example, the system console device could be assigned to a CRT and a modem at the same time. In this way it would be unnecessary for your program to output a character to both devices because the BIOS logic has enough information passed to it to do it automatically.

Each segment of the CHRTBL (Table 2, page 116) is eight bytes long. The first six bytes contain a physical device name of your choice. The next byte is used to describe whether this device can accept input or output or both and additionally whether baud rate is software selectable. If any communications protocol is necessary, a bit is set for that also. The last byte contains the baud rate for serial devices. Obviously, this field would have little use if your hardware is unable to handle software baud rate selection. Up to a maximum of 12 table segments can be defined; in all but a very large system that will be more than enough.

Using this facility is a two-step process. First, issue a BDOS function 20 that returns the CHRTBL table address in the HL register pair. Next, use BDOS function 49 to gain access to the section of the system control block that contains the vector for the desired device. The returned vector is then used to create a pointer into the CHRTBL.

This has been a very brief description of what could be a very important addition to your system, especially if you use I/O redirection heavily. **DDJ**

OF INTEREST

by Michael Wiesenberg

New Multibus Standard

Engineers, designers, and other electronics industry cognoscenti are all familiar with Intel's standard microprocessor bus, the IEEE 796 Multibus. Now **Intel** announces **Multibus II**, the new architecture for advanced 8-, 16-, and 32-bit high-performance microprocessor systems. Thirteen companies have joined together to develop the specifications for the new bus: Advanced Micro Devices, Compagnie des Machines Bull, Dataindustrier AB, Hewlett-Packard, Intel, International Computer Ltd., Intersil Systems, Matra, Mupac Corp., Nixdorf Computer Corp., Siemens AG, Tektronix, and Zilog. According to Intel, "the new development and review effort is intended to accelerate the process of industry standardization by collecting the views of manufacturers and users. Intel expects to publish Multibus II specifications in the second half of this year. The company will also provide Multibus II review materials to the recently formed Multibus Manufacturers' Group, an independent organization of companies supplying Multibus-compatible products and service. . . . These modular, processor-independent bus structures offer capabilities that expand system bandwidth, enhance multiprocessing, increase system reliability, and improve ease of use."

Standardization and *processor-independent*. Those are the key words that may mean more compatibility among future microcomputers. Let's hope so! **Reader Service No. 101.**

Apple, Meet CP/M; CP/M, Meet Apple

System One, from **Extra Computer Corporation**, is a dual processor (Z80/6502) system that comes with both CP/M 2.2 and Apple DOS, Applesoft, Perfect Writer, Perfect Calc, and a games package. You also get 64K RAM, 8K system ROM, 16K user ROM, seven peripheral slots compatible with Apple, 40/80 column display, high and low res graphics, both NTSC and RF modulated video outputs, 110/220 power supply, sculptured keys, upper and lower case, shift lock, and numeric key pad. The case is metal, which is supposed to provide RF interference shielding. And for all this you pay only $795. **Reader Service No. 105.**

S-100 Process Control

Smartboard RC-100 from **Automatic Micro** controls and checks the states of up to 16 remote relay switches, to turn on and off and monitor lighting, appliances, electrically controlled heating and cooling devices, or entire electrical circuits. You get optically isolated I/O, zero voltage switching for extended relay contact life, and port or memory-mapped addressing. You also get an on-board 8035 CPU and four status/command registers. Use it with GE RR8 remote control relays (latching relays able to switch loads up to 20 amperes, with states that are unaffected by power outages). You can control all the board's features from your S-100 computer with a BASIC program, or an external 2716 or 2732 EPROM permits custom programming. For $495 you get the board, documentation, and application software examples. **Reader Service No. 109.**

A Real Black Box

Black Box SAM+ Centronics, a Status Activity Monitor from **Black Box Corporation**, reconfigures, patches, monitors, and tests all active interface leads on Centronics-compatible parallel equipment. This so-called *break-out-box* gives direct access to all active signal and ground leads of the interface to ease trouble-shooting. Power comes from the interface to which the unit attaches with two female connectors and a 12-inch male-to-male ribbon cable. The Black Box permanently monitors Data Strobe, Bits 1 through 8, Acknowledge, Busy, Paper Empty, Select, OSCXT, Input Prime, and Fault. Other leads can be tested for ground presence and pulse by patching into four spare LEDs. You also get negative and positive pulse detector circuits to locate and identify intermittent pulses, and two inverters to invert Strobe or Acknowledge lines. It costs $225, and you get a one-year warranty. Black Box will also send you a free catalog. **Reader Service No. 103.**

Color It Blue

You don't have to pay an upper extremity and a lower appendage for a full-screen editor with multiple windowing for the IBM PC, XT, and compatible clones. **Blue**, from **Symmetric Software**, uses full color (and also works fine in monochrome), simultaneously edits multiple files, offers on-line help, has macros, formats with embedded commands, does all the stuff standard with most word processors, is not copy protected, and costs $150, or just the manual for $35, or a demo disk for $10. **Reader Service No. 111.**

Color It 8

Eight-pen plotters are coming down in price. **Strobe's M260**, previously introduced as a six pen plotter, now has eight, which it changes automatically under program control for unattended multicolor output. Resolution is 500 steps per inch. You also get an RS232C interface, 1K buffer, upper-case and lower-case character sets, including foreign language characters, and graphics software for bar, pie, and line graphs. All this for $995. You can get the Enhanced Business Graphics software package for plotting graphs from VisiCalc and SuperCalc files and from data entered at the keyboard. Strobe says that over 30 other popular graphics packages from other companies also support Strobe plotters, and most of these are adding drivers for the M260. Strobe has also lowered the prices on its M100 and M200 single-pen plotters to $595 and $695, respectively. These plotters and the graphics software run on most popular microcomputer operating systems. **Reader Service No. 113.**

Hard-Wired Desks

Tired of bulky, cumbersome computer desks and tables that *look* like computer desks and tables? You probably don't even *feel* comfortable coding on what the industry terms a workstation. **Riverside Furniture** has introduced four new computer desks that look like traditional furniture. One of these beauties is a real roll-top. All are made of real wood, with pull-out assemblies for printers, paper, and equipment, interior storage for programs and disks, and totally integrated electrical systems with surge protectors and circuit breakers. Prices range from $330 to $1760. **Reader Service No. 115.**

Can You Get a Chocolate MULTED?

Search, edit, print, and manipulate text in multiple files without having to operate individually on each with **MULTED** from **Compu-Draw**. You can specify equivalence of upper and lower case, "wild card" searches, and searches with column position specified. You can search for blank lines, or lines *not* containing specified text. You can also specify conditional editing, for example, changing all occurrences of "Dobbs" to "Dobb's" only if the words "Dr." and "Journal" also appear on the same line. MULTED automatically backs up files as you work on them, alternating drives. MULTED runs on CP/M, in many disk formats, and costs $59. **Reader Service No. 117.**

Super C Superseded

The **SuperSoft C Compiler** has been updated with a new release syntactically compatible with Unix version 7C, and adding long integer and double floating point functions, plus an extensive library of functions. This multipass compiler produces optimized code, obviating the necessity for assembly code in most applications. Source code can easily be ported between Unix and SuperSoft versions of C. $275 for CP/M-80, and $500 for CP/M-86 and MS-DOS. If you already have SuperSoft C, you can get an upgrade for *$100*. **Reader Service No. 119.**

EBZUG, PUG, SBAZUG

If you've bought one of those cute little Timex computers, and can't get it to work right, what do you do? It's hard to find someone to service a computer that costs under $100, but don't junk it. Go to a users' group meeting. You'll find long tables with electrical outlets. Everyone brings and sets up a system, and you can find plenty of people to help you debug your problems. You'll also get hints on new and efficient programming techniques, demos of peripherals that you never thought you could hook up to such a tiny machine, bargain software, tips on where to buy equipment at the lowest prices, and probably make new friendships. In the Bay Area, the **East Bay Timex/Sinclair Users' Group** (EBZUG) meets the third Thursday of each month at 7:30 p.m. at the Berkeley West Branch Library near the corner of University and San Pablo. Yearly dues are $15 for full newsletter and library privileges, $10 for the newsletter only, and $8 for student (through high school). The **Peninsula Users' Group** (PUG) meets the third Sunday of each month at 1 p.m. at Peninsula Hospital, 1783 El Camino, Burlingame. The **South Bay Area Timex/Sinclair Users' Group** (SBAZUG) meets from 7 to 10 p.m. the last Tuesday of each month at Dysan's north entrance by the loading dock, 5401 Patrick Henry Dr., Santa Clara. The three together publish the newsletter, *Timelinez*. **Reader Service No. 127.**

Free Turtle Alphabet

To introduce you to their newsletter, **The National Logo Exchange** is offering to readers of *DDJ* a free Logo listing that draws alphabet letters in any size or orientation. Send SASE to DDJ ABC's, c/o The National Logo Exchange, Box 5341, Charlottesville, VA 22905. The newsletter, for teachers who use Logo in the classroom, is published September through May, and costs $25 per year in the U.S., Canada, and Mexico. Elsewhere, add $5. **Reader Service No. 121.**

Two-Bits' Worth

For your Timex-Sinclair 1000, "at last," they say, "computer games designed especially for women." **Working Mother's Dilemma** from **2-Bit Software** is a package of four games for the **Harried Housewife** (one of the games), including **Carfool**, testing your ability to use the least gas while errand-running about town, and "dazed search for your car in the parking lot of Shopping Mall." $14.95 each (Californians add 6% tax). **Reader Service No. 125.**

Time Works on Commodore

Who says you can't use your computer to balance your checkbook? If you have a Commodore 64, you can get **Electronic Checkbook** from **Timeworks**, for recording, sorting, and balancing your checkbook; or an educational game, **Dungeon of the Algebra Dragon**; or **Programming Kit I**, a teacher of BASIC and game and program design. Each program costs $24.95. **Reader Service No. 123.**

$800 Daisy

The **DWP-210 Daisy Wheel Printer** from **Radio Shack** is code-compatible with Scripsit, has a standard parallel interface and serial interface compatible with the Color Computer, prints 10- or 12-pitch or proportional spacing, and comes with carbon ribbon and a 10-pitch wheel. Radio Shack claims 200 words per minute for the printer, which sounds impressive, until you realize that most printers are rated in characters per second. Let's see, the average word has about six letters, and 1200 characters divided by 60 seconds is, yes, 20 cps. You can also get the **DWP Bidirectional Tractor Feed Kit** for $149.95. They neglected to tell us if you can use the printer with other computers, so you should check. **Reader Service No. 107.**

DDJ

Letters

(Continued from page 10)

2a+b is divisible by 7. The rule for casting out nines was known to Avicenna (980-1037). Leonardo of Pisa (Fibonacci) indicated the tests for 7 and 11. The problem was "solved" (the quotes are necessary for reasons suggested below) in sweeping generality as early as Blaise Pascal, and is discussed at some length by Gauss in the *Disquisitiones Arithmeticae.*

I am sure that Gordon did a reasonable literature search; but I am also sure that the texts consulted were oriented toward the practical needs of persons doing computation – Abramowitz and Stegun, the CRC table, college algebra texts... that sort of thing. Except for the easiest cases (divisibility by 2, 3, 5, 11, and certain multiples of these) the digit-based divisibility checks are not of any practical computational use because of the labor they involve. This means quite simply that the checks will not be discussed in books oriented toward computation. However, the techniques that Gordon described are standard fare in any undergraduate number theory class, and are usually discussed there in full generality.

Even though the techniques are not computationally useful, they are mathematically interesting, and do have some specialized computational uses. For interested readers of *Dr. Dobb's Journal,* I will try to outline the ideas involved.

Following the usual notation, let us write $a \equiv b \pmod p$ [read: "a is congruent to b modulo p"] if a and b leave the same remainder upon division by p, or equivalently if p divides (a–b) evenly. The smallest non-negative integer c such that $c \equiv a \pmod p$ will be called the *residue* of a (mod p).

In order to keep the discussion simple, assume that p is a prime. Then it is possible to show that if a is a number *not divisible by p,*

$$a^r \equiv a^s \pmod p$$

if the quantity (r–s) is divisible by (p–1) [this is the "little theorem" of Fermat – the less said about the "great theorem," the better]. The condition, although sufficient, is not necessary:

$$2^2 \equiv 2^5 \pmod 7.$$

When the condition *is* necessary – i.e., when any consecutive (p–1) powers of a are distinct (mod p) – then a is called a *primitive root* (mod p). Primitive roots (mod p) always exist for any p, but there is no procedure (apart from trial and error) that will allow us to say whether a given number a is a primitive root (mod p) for a given p. This last problem – the "primitive root problem" – is extremely difficult, and has defied the best efforts of many of the most talented mathematicians of the last three centuries. It is not even known whether any given number, say 2 or 10 (the two bases we are most likely to be interested in), is a primitive root for an infinite number of primes.

For any given a, primitive root or not, we can define a number called the *period.* This is the number of consecutive powers of a that one must compute before the residues start repeating. For example, the powers of 2 (mod 7) are 1, 2, 4, 1, 2, 4, 1, ... so the period of 2 (mod 7) is 3. A primitive root has period (p–1) by definition.

Now, let us consider the divisibility question. For any positive integer $b > 1$, a non-negative integer a has a unique representation,

$$a = \sum_{k=0}^{n} a_k b^k,$$

where the a are non-negative integers between 0 and (b–1), and the high-order coefficient isn't zero. The base-b representation of a is then just $a_n a_{n-1} \ldots a_1 a_0$. A digit-based divisibility test for the prime p (I will not apologize for using the word digit) seeks necessary and sufficient conditions on the a_k so that p will divide a, or in the language of congruences,

$$\sum_{k=0}^{n} a_k b^k \equiv 0 \pmod p.$$

But from the last congruence, it is clear that we need only consider the residues of powers of b. These will repeat after some number of times (the period), so we end up with a criterion of the form

$$\sum_{k=0}^{n} a_k B_k \equiv 0 \pmod p,$$

where the B_k are the residues of the corresponding powers of b.

An example in base 10 should make this all clear. Modulo 11, the powers of 10 are 1, 10, 1, 10, 1, 10, ... Thus, the criterion for a base-10 integer to be divisible by 11 is

$$\sum_{k=0}^{n} a_k B_k \equiv 0 \pmod{11},$$

where $B_k = 1$ for even k, and $B_k = 10$ for odd k.

But the expression in the B_k gives a new number a′, which it is easy to see must be smaller than a. If the original a was divisible by 11 then so is a′ (that's what $\equiv 0$ means), so we can repeat the procedure, giving us another number a″ that is smaller than a′ and that is also divisible by 11. We can continue in this way until finally we end up with a number smaller than 11 in absolute value. But the only number smaller than 11 in absolute value that is divisible by 11 is zero – thus the congruence turns into an equality.

We can simplify the criterion by noting that 10 is congruent to –1 (mod 11), so we could have used –1 in place of 10 to give us our new numbers a′, a″, ... This says that if we start at the units digit and subtract the tens digit, add the hundreds digit, subtract the thousands digit, and so on – and then take absolute values, and repeat the procedure as often as necessary to get a number smaller than 11 in absolute value – the original number will be divisible by 11 if and only if our final anwer is 0. This is the familiar rule for casting out elevens.

Readers might find it instructive to try the same reasoning on divisibility by 3 (casting out threes). The rule for casting out nines is similar, although a slight modification of the theory is needed since 9 is not prime.

The problem that comes up in the general case is that it is not known what the period of 10 (or 2, for machine arithmetic in binary) is for arbitrary primes p. This is for all practical purposes the primitive root problem for b = 10 (or b = 2) and, as was said before, that problem is quite intractable. This complicates the theoretical analysis, although it does not interfere with setting up a divisibility criterion for any given p as long as we are willing to compute all the residues of at most (p–1) powers of 10 (or whatever). Even so, a little experimentation will show that the resulting criteria are not necessarily convenient (try divisibility by 17 in base 10).

Persons interested in pursuing this subject further may consult any standard undergraduate number theory text.

Yours truly,
John Kahila
217 Rock Harbor Road
Orleans, MA 02653

An Address Correction for ACG-NJ

Attn: Editor

We were happy that you gave a review of PISTOL – Portably Implemented STack Oriented Language – in your February issue. I would like to correct the address of the Amateur Computer Group of New Jersey (ACG-NJ) from which potential users of this public domain software may obtain an 8-inch diskette. The diskette is available from the Special Interest Group in Microcomputing (SIG/M) of the ACG-NJ, which is located at: SIG/M, P.O. Box 97, Iselin, NJ 08830. PISTOL currently exists as SIG/M Volume 059 release 1.0 and SIG/M Volume 114 as release 2.0. The cost of the diskette is $6.00 ($5.00 plus $1.00 shipping). International costs are an additional $8.00 (U.S. currency) per order.

Please be patient when ordering since this is a voluntary organization dedicated to making available the best CP/M and PCDOS/MSDOS public domain software.

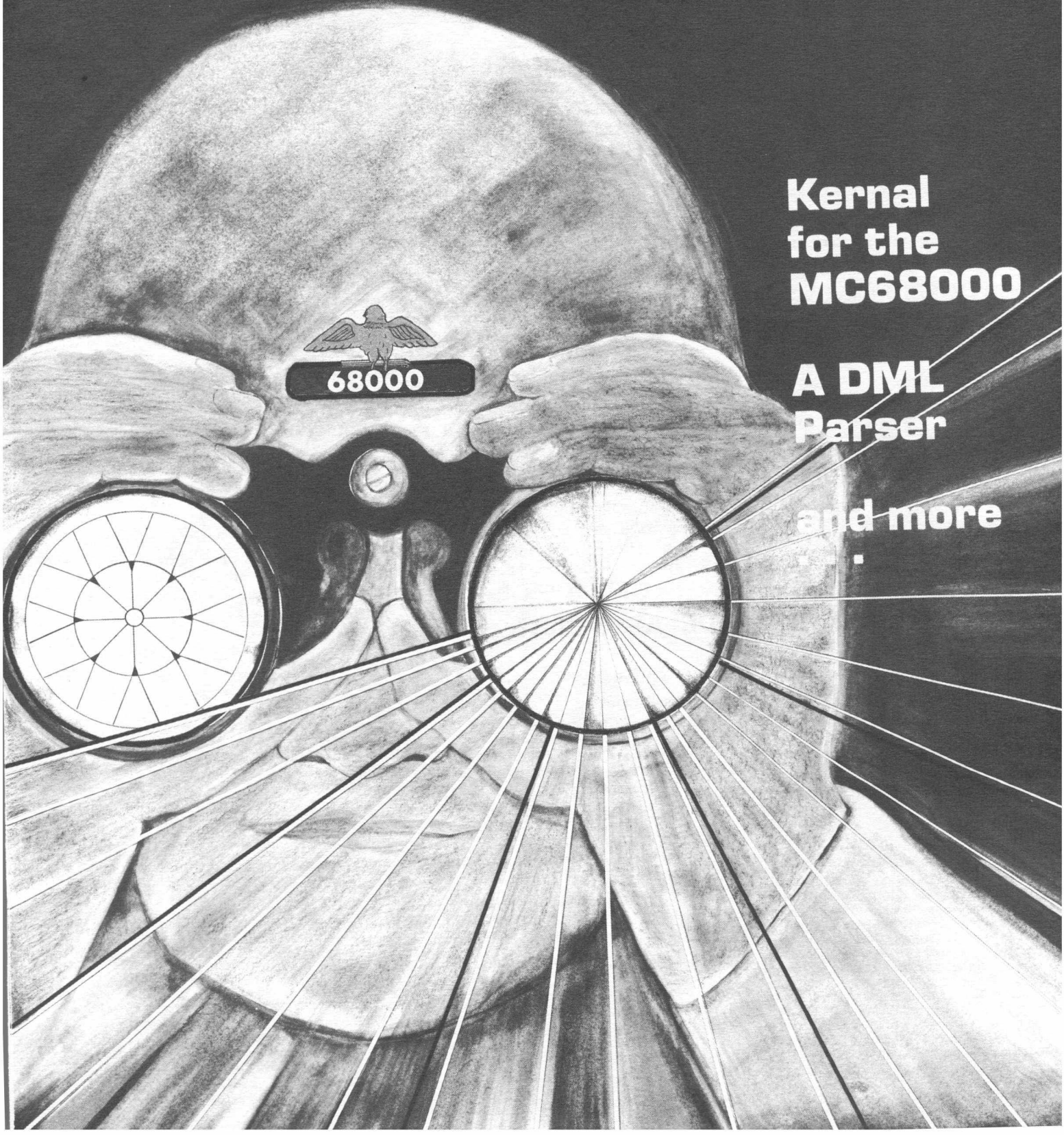
Sincerely,
William S. Chin
Chairman of SIG/M
P. O. Box 97
Iselin, NJ 08830

DDJ

#85 November 1983

Dr. Dobb's Journal

For Users of Small Computer Systems



Kernal for the MC68000

A DML Parser

and more

Publisher – Jane Nissen Laidley
Editor – Reynold Wiggins
Managing Editor – Craig LaGrow
Contributing Editors –
Robert Blum, Dave Caulkins, Dave Cortesi, Ray Duncan, Michael Wiesenberg
Marketing Director – Beatrice Blatteis
Marketing Assistants –
Sally Brenton, Ed Gueble
Advertising Director – Carl Landau
Advertising Sales –
Alice Hinton, Walter Andrzejewski
Accounting Manager – Gloria Romanoff
Accounting Assistants –
Peter Koral, Chris Neylan
Circulation – Terri Pond
Staff Support – Billie Greenwood
Production Manager – Barbara Ruzgerian
Production Assistants –
Shelley Rae Doeden, Alida Hinton, Tom Martin
Typesetter – Paula Fairchild
Cover Illustration – Barbara Ruzgerian

Copyright © 1983 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Christine Bell, W. D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, John W. Campbell, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Thomas Davis, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III, Infoworld, Stan Veit, Western Material Control, S. P. Kennedy, John Hatch, Richard Jorgensen, John Boak. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

Dr. Dobb's Journal
For Users of Small Computer Systems

November 1983
Volume 8, Issue 11

CONTENTS

ARTICLES

20 A Kernal for the MC 68000
by Steve Passe
Having obtained a 68000 system, the author found himself without sof'ware to run on his new friend. After punching out a cross-assembler in C for software development, he decided to experiment with multiple tasking. The result was this kernal, which distributes CPU time among concurrently running processes.

46 A DML Parser
by David Harry
How many times have you finished all but the user interface for that great new utility and decided it's just too complicated to develop a sophisticated command processor? DML is an attempt to provide a simple, powerful parsing system for microcomputer users which is designed to be imbedded in any application program that requires sophisticated syntax recognition.

80 Toward a More Writable Forth Syntax
by Harvey Glass
While powerful and efficient, Forth is often criticized for creating unnecessary programmer obstacles. The goal of the Forth extensions discussed here was to preserve the power and flexibility of the language while providing a simpler and more readable notation. The resulting code is not unlike a postfix dialect of Pascal or Algol.

86 Simple Graphics for Printer
by Wil Baden
The author presents a classic algorithm for plotting lines on a terminal or line printer. Though the Pascal listing provided is designed to produce low resolution graphics, the algorithm may also be the basis for much higher resolution routines.

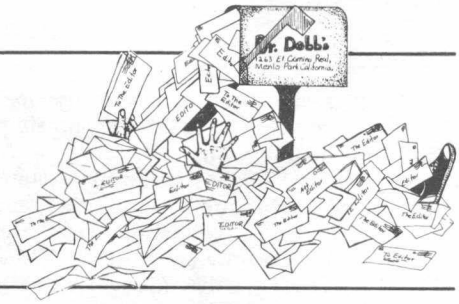
DEPARTMENTS

8 Letters
9 Editorial
10 Dr. Dobb's Clinic
"Help wanted" for PC and P-system, Double Amplification, and the Intern's search for a good system programming language
92 Software Reviews
98 Book Reviews
102 CP/M Exchange
8080 to Z80 Program Conversion, Hidden Treasures *Revisited*, *CP/M Plus* DPB Macro Fix, and Quicker Submit File Truncation
110 16-Bit Software Toolbox
Some PC items, Floating-Point Benchmarks, and an MSDOS COM File Loader
118 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to People's Computer Company, Box E, Menlo Park, CA 94025 · 415/323-3111

ISSN 0278-6508

LETTERS



Getting Double Density on your Xerox 820

Dear Editor,

I would like to pass along a little information that I have learned with regard to my Xerox 820 Version One.

I purchased the machine in early 1982 and have been doing mainly Z80 assembly language programs for my own use and pleasure. I have been constantly irritated by the limitation of 81K of useable space per minifloppy drive and have on occasion thought of purchasing two more drives, although that would have been very difficult for me to do as the cost is too much and the Computerland from whom I bought the Xerox no longer carries Xerox.

Well anyway, also being distraught by the lack of software for it, I bought a cheap Osborne game to see if I could read it onto the Xerox – reasoning that if Osborne can read Xerox disks, then Xerox (with sufficient manipulation) could read Osborne disks.

It did not work. Doing everything I could with the disk attribute tables effected no read. I kept receiving a "data overrun" error from BIOS (not BDOS).

Well, the Xerox has a monitor that is external to CP/M and I eventually tried to read sectors off the Osborne with it (it has a read command); I got the same error, but it turns out it actually was reading in 128 bytes of data from the Osborne diskette.

To make a long story short, I found a location in the monitor RAM that was set to a %80 (hex) and for the heck of it changed it to zero, and lo and behold, the Xerox monitor happily read 256 bytes of data for a sector read off of the Osborne disk (it seems the Osborne has 256 byte physical sectors).

The place in question is OFF65H or %FF65, which might differ on other 820-I's. I found that I could change it in CP/M and have it read also. Well, what this all means is that I could double the density of the Xerox without spending any money at all (although I do count my time as money). I changed the disk formatter program with DDT to alter this location and ended up with Xerox diskettes that have a capacity of 184K bytes of storage.

There is a catch, though. I will have to write a new BIOS and write some utility routines to copy from the old to the new. (The Xerox format is now set to 41 tracks, 18-256 byte sectors/track.) I have written all but the BIOS disk write routines – those being harder than the read routines – as BDOS wants 128 bytes and the hardware supports 256 bytes and I might end up having to do a read in order to effect a write.

Well, I hope that someone out there might find this information of use. I suspected that it might be possible as the 820-I that I own has SA400L drives, which sometimes are described as double-density drives.

Thomas Kellar
3104 Hassler St.
Dayton, OH

Moaning and Groaning About CP/M

Dear Dr. Dobb,

While Bill Potter is "Moaning and Groaning about MS-DOS" (July 1983 *DDJ*, Letters, page 10), it is interesting that he put some changes in CP/M's SUBMIT and XSUB by disassembling them and (apparently) adding some machine language patches. I didn't know that modifying CP/M is so easy. (It is also interesting that Mr. Potter is spending considerable time improving CP/M's utilities, but does not see fit to "moan and groan" about CP/M.)

I believe that the code which implements the MS-DOS batch facility is in the module COMMAND.COM, rather than in

Dr. Dobb's in Istanbul

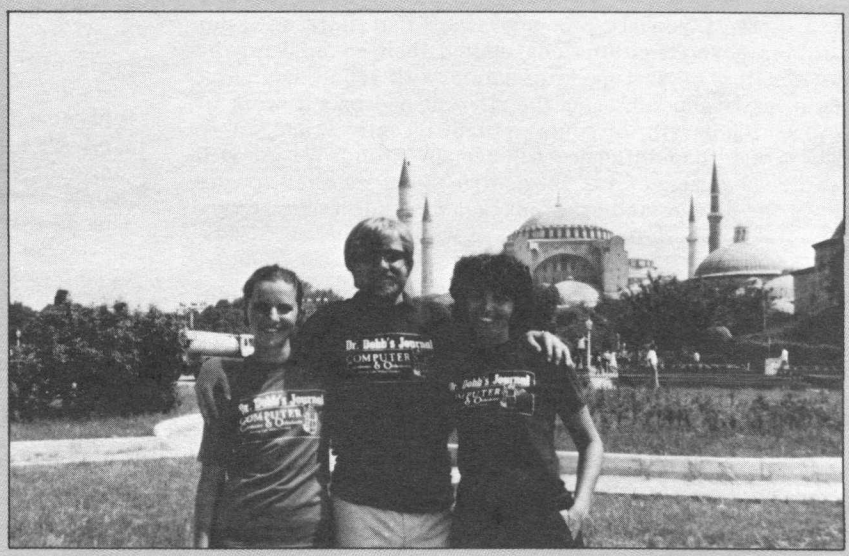
Dear Dr. Dobbs,

Our official Dr. Dobb's T-shirts drew many appreciative stares on our recent visit to Istanbul, Turkey. That's the Blue Mosque behind us (see photo at right).

The microcomputer revolution has yet to hit Istanbul (video-rental clubs are the current craze) but we did meet some young Turks who are in the vanguard of the micro movement and who devour everything in print on the subject and revere *Dr. Dobb's Journal.*

Soon, the world.

Your faithful reader,
Steve Rank
1260 Monument Blvd.
Concord, CA 94518



the DOS itself. Modification by disassembly and patching might be possible, though probably rather difficult.

MS-DOS 2.0 has conditional statements and looping in its batch files, and also supports I/O redirection. This latter feature allows a program to read its keyboard input from a disk file (and/or write its CRT output to a disk file) and is one of the major features missing from CP/M. Thus a command in a batch file can specify a disk file to be used in place of the keyboard input, which gives the same capability as CP/M's XSUB. Note that the use of a separate file for input prevents a command from reading, as input, lines that are intended to be commands.

I find it a little strange to be defending MS-DOS, since it is basically a CP/M derivative, and I regard CP/M to be a poorly designed and very primitive operating system. At least MS-DOS is not an exact copy. My own major "moan and groan" about MS-DOS is its lack of a DUMP command.

Sincerely,
Jim Howell
5472 Playa Del Rey
San Jose, CA 95123

NEC PC8001/Epson MP80 User Groups Anyone?

Dear Editor:

Are there many NEC PC8001/Epson MP80 users out there who:

- Have unresolved questions/cannot get English manuals/have useful information not covered in the English manuals?
- Know of individuals/companies offering programs – in NEC 5¼-inch disk format – to run under CP/M or other DOS?
- Know of companies offering NEC PC compatible cards – or would like to know of such companies?
- Know of any active user groups, or would like to form a users' group?

If there are enough people with the same problem, maybe I can send Dr. Dobbs a (translated) answer.

Yours truly,
Keith S. Wilkinson
CPO Box 1748
Tokyo, JAPAN 100-91

A Cross Assembler for the MC68000

Dear Sir:

The Software Review article by Steve Newberry on page 108 of your August issue states that there are no cross assemblers for the MC68000 which run on the 8080.

We would like to draw your attention, and that of your readers, to the fact that farbware has been offering just such an assembler since June of 1983. This assembler is compatible with the assembler provided by Motorola for their development systems and offers conditional assembly, macros, and structured programming facilities.

Farbware intends to offer a series of cross assemblers for 16-bit microprocessors such as the 16032 in the future.

We have been subscribing to *Dr. Dobb's* for some years now, and have every issue back to the first. The quality of your publication is excellent.

Sincerely,
David Farb
President
Farbware
1329 Gregory
Wilmette, IL 60091 DDJ

EDITORIAL

DDJ Pays Authors!

Truth truly *is* stranger than fiction. Starting this month, *DDJ* will be paying for articles!

Most of you are probably aware that, until now, our authors have generously contributed their material for the satisfaction of sharing information with fellow computer enthusiasts and of seeing the effect it had on the work of others. These are, of course, potent incentives, and we assume that their influence will remain strong. We are well aware, however, of the substantial effort required to produce the articles and programs that we publish. We are very grateful to all those who have so willingly shared their labors and expertise with their colleagues and fellow readers.

Because we know what it takes to produce a manuscript, we have wanted to pay authors for some time. Nonprofit status being the mixed blessing that it is, we were forced to learn the virtues of patience. Now we are moving, and we expect to have the entire payment program firmly in place by the New Year.

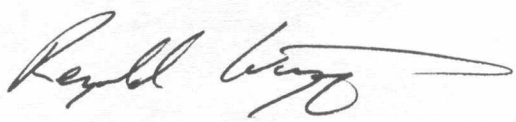
We would like to thank all our supporters and advertisers who have helped make this possible. And we are pleased to be able to express our thanks to our authors in more concrete terms. Give us a call or drop us a line to find out the details.

* * *

We have scheduled the February 1984 issue as our telecommunications specialty issue. For those interested in submitting material, some of the topics that we are considering are local area networking, security in telecommunications, protocols, and micro-to-mainframe hookups. This list is certainly suggestive rather than exhaustive. If you have a topic that you are interested in writing about, let us know soon. We have set the copy deadline at November 10, 1983. Since the timeframe on this one is pretty tight, interested people should contact us at (415) 323-3111 for details. This part of the field is growing quickly, and we expect this special issue to provide some timely information on several of its facets.

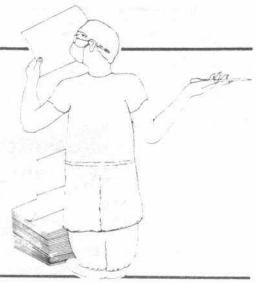
* * *

In closing, we would like to thank all those who responded to our request in August for technical referees. The turnout was very good and we are well on our way to having the review board completely set up.



Reynold Wiggins

DR. DOBB'S CLINIC



by D. E. Cortesi, Resident Intern

P-System Help Wanted

At last, a visitor to the Clinic! Dwight Irving, of the University of Idaho, sent us the following plea.

"I have been trying to install a UCSD p-System Version IV.0 from Softech Microsystems and have encountered several problems. The computer has a Z80 processor, 64K RAM, a CompuPro Disk 1 controller, and two 8-inch drives. The operating system is CP/M 2.2 for which CompuPro supplied its GBBIOS.

"I tried to use the PASBOOT program to boot up the CP/M adaptable system. Since the p-System disks were single density it was necessary to recalibrate [*re-log-in? DEC*] the drives before trying to read the secondary booter; it refused to be read anyway.

"I next obtained a working p-System disk from another computer. I changed the PASBOOT program to include disk routines from the CompuPro program. The attempt to boot was only partly successful; the secondary booter was read in but an error "Can't find SYSTEM.PASCAL" occurred. The same disk will boot up with another computer; the skew values I used are correct for this disk. Also, no error occurred during the BIOS disk routines.

"Both CompuPro and Softech tell me the problem is a nonstandard BIOS and that I will have to use the full adaptable system. I would appreciate any information that would help in writing the SBIOS for the Disk 1 Controller or on how to fix it so that I could use the CP/M adaptable system."

We don't know zip about the p-System, but surely some reader does. What are the steps by which the p-System comes up from a CP/M base, and where do you twiddle the hardware-dependent parts of it? All letters will be passed on to Dwight and summarized here.

Also PC Help

W. B. Koehler, of Foster City, California, also has a question. "I have been told that on the IBM PC one can manage cursor control via a BASIC Print statement using the CHR$ function. Apparently the string begins with the ESC (027) character followed by some screen positioning scheme. Can you find out the truth of this matter?"

That's certainly how you manage the IBM printer functions, and CP/M-86 supports cursor control using escape sequences. But if it's true of MSDOS, it must be something added in DOS 2.0. Readers?

Double Amplification

Allan L. Behler, Jr., of Newport, North Carolina, writes at length on the problems of numeric precision in BASIC. He warns that PRINT USING, although it may produce a correct display of a slightly inaccurate number, has no effect on the variable being displayed. "I.E., if you print out 1000 double-precision numbers such as 134.12 with

```
PRINT USING "#####.##"
```

the individual numbers may appear correct, but their total in storage can be off by as much as 0.06. I know because the first business software I wrote did just this, and caused some ugly phone calls."

Then Allan leads up to an interesting problem and a useful solution through a series of experiments. Try them out yourself to see if you get the same results. Begin in immediate execution.

```
DEFDBL A
A = 134.12
PRINT A
134.1199951171875
```

Aha! Now round it up.

```
A1 = INT( A*100 +0.51 )
PRINT A
13412
```

A is still carried as double-precision floating point, but its value is integral, and a correct representation of the desired number (times 100).

```
A2 = A1/100
PRINT A2
134.12
```

As if by magic, the untruncated value appears. Allan next tries it as a function.

```
10 DEF FNRD(X) =INT(X*100+0.51)
20 DEFDBL A
30 A=134.12          : PRINT A
40 A1=FNRD(A)        : PRINT A1
50 A2=A1/100         : PRINT A2
RUN
134.1199951171875
13412
134.12
```

Allan notes that you could use this function to force all numbers to double-precision integral values, process them in that format, and then print them, divided by 100, with PRINT USING for rounding. However, let's try the whole job in one function.

```
NEW
10 DEF FNRD(X)=INT(X*100+0.51)/100
20 DEFDBL A
30 A=134.12          : PRINT A
40 A=FNRD(A)         ; PRINT A
RUN
134.1199951171875
134.1199951171875
```

"Now," says Allan, "I have to admit that this had me going for a year. The function would work fine for single precision but not for double precision. Then a friend solved it for me in about 12 seconds (you know the type of friend I mean)."

```
NEW
5  Z#=100
10 DEF FNRD(X)=INT(X*100+0.51)/Z#
20 DEFDBL A
30 A=134.12          : PRINT A
40 A=FNRD(A)         : PRINT A
RUN
134.1199951171875
134.12
```

"The use of a double-precision divisor of 100 makes the funtion work for both single- and double-precision numbers."

In Search Of . . .

We ain't assembly language freaks. We don't *like* assembly language. It's a tedious, error-prone medium for expressing one's ideas, and its use ensures nonportable code. You can ameliorate the tedium and error-proneness with a good library of standard subroutines, but there's nothing you can do about the nonportability. And that, when the market for software is fragmented by so many incompatible systems, is a killer.

So what we, and probably every programmer, want is a good, medium-level, systems programming language. A language needs two characteristics to be "good" for this use. It has to be a good notation for the expression of algorithms, and it has to compile to object modules that are, at worst, twice as large and half as fast as well-written assembly language.

It isn't impossible, you know. For some years we made daily use of PL/S, an IBM proprietary language for systems programming on the System/370 series. PL/S met both criteria. It was a tolerably expressive notation (basically PL/I with extensions); and the compiler did extensive global optimization, so it built object code that was not an affront to the soul.

But that is all nostalgia now; we want a systems programming language for personal computers. As far as we can tell, there are none.

For a while, we had hopes for Pascal. It isn't a great notation for systems programming, but it can be stretched to do the job. Furthermore, its syntax lends itself to optimal object code. Pascal is designed so that the writer has to tell the compiler all sorts of things about the way variables are used. An optimizing compiler could use that information to produce good code.

Unfortunately, Pascal's syntax is also designed to permit a simple, one-pass, recursive-descent compiler – and that is what all the personal compilers are. The techniques of global optimization have been well-known for a decade; Pascal's syntax is admirably suited to global optimization: yet we don't have any optimizing Pascals. They all produce code that is dumb, dumb, dumb.

Looking to C

All the buzz is that C is the answer to our prayers, an expressive notation that compiles to efficient code; so we recently gave it a try. Keith Coy, C guru and president of PicoNet, a local CP/M group, recommends Aztec C, so that's what we bought. It's a clean implementation, he says, closely compatible with Unix C. It is all those things. We don't recall he said it was fast, and it isn't. Most comparative benchmarks put the Aztec implementation in the second rank for object size and speed; so we figured that if our programs came out a little bit slow, we could move to a better compiler without changing languages. In the meantime, reliability and standardization would be nice to have.

So we wrote our first C programs. The first nontrivial one was a filter that accepted an Aztec C link map, sorted it into ascending order by addresses, and wrote it out. You can read the final version in Listing One (page 14).

The first thing we learned about C is that it takes some learning. Take the use of the "item" structure in Listing One, for instance. It is not intuitively obvious that

```
p -> addr
```

has the value of an unsigned integer, while the expressions

```
p -> id
&(p -> id)
```

are identical and yield the *address* of a character string. These discoveries were made in the course of two hours' work getting scanf to do what we wanted it to do. At one point we peeked into stdio.h, saw this,

```
int (*SYSVEC[MAXVEC])()
```

and were sidetracked for 90 minutes with Kernighan and Ritchie, working out what it meant. (It means SYSVEC is an array of MAXVEC elements, each a pointer to a function that returns an integer. You read it outward from the center in a spiral.)

The pristine eloquence of our original program has been somewhat distorted. The constant IGNORAD was introduced because, for unknown reasons, scanf returns one line of garbage before it discovers EOF. We're probably doing something wrong, or maybe we found a bug. That's the trouble with new languages; you can't tell your own mistakes from the compiler's.

Then there was the baffling system hang, which persisted until we renamed our output function from "write" to "display." There's a runtime function named "write," it seems, which wasn't linked because ours was present. Is that a bug? Or would an experienced C user know not to use the name "write"?

Warming the Molasses

In the first version, output was produced with a printf call very like the scanf call that gets the input. We got it working, and it was slow. Not merely slow compared to assembly language, but slow as molasses. Given its own link map (200 lines, about 2500 bytes), it took 68 seconds to complete. That's intolerable in a filter-type program, so we stuck the message lines in "main()" to find out what was taking all the time. It broke down this way:

- load from disk, set up in and out files: 5 seconds
- reading: 7 seconds
- sorting: 1.2 seconds
- writing: 55 seconds

The Quicksort routine was quick enough. Reading data at 350 bytes/second was slow, but the real boggler was writing. What in the world afflicted output?

To find out, we used the SID debugger with its HIST utility routine. This useful combination is described at length in the SID manual. HIST collects a count of instruction addresses during a trace and displays them as a histogram of execution density. Listing Two (page 16) shows a typical display. It is a coarse measure of execution density over the entire span of the object module. Listing Three (page 18) is the result of zooming in on a smaller span of code. It has a fine enough grain that individual runtime routines can be picked out.

When you look at Listings Two and Three, bear in mind that the code that was generated from Listing One ends at 0x0598. All the rest is from the runtime library.

HIST showed a particularly hot spot in a routine called "ldiv." By examining the source of the runtime code, we discovered that ldiv was the long-integer divide routine. Further prowling revealed that printf used a long-integer divide to generate each digit of an output integer. Every execution of

```
printf("%04x %s",p -> addr,p -> id)
```

caused four long divides. That's why we wrote the routine named "xprint" in Listing One. Its use eliminated the long divides, and that took 8 seconds off the output time, bringing it from 55 seconds to 47.

We found, however, that the cold molasses was actually in the disk output code. When stdout was not diverted to a file, but rather allowed to come to the screen, the output phase took only 5 seconds, not 47 (we had previously shaved off 8 seconds, reducing it from 13 to 5 seconds). Further zooms with HIST turned up a broad hot-spot encompassing the library routines for sequential disk output. These are clearly less than optimal. We don't know what is wrong with them; they may be calling the BDOS to read back the current sector on every byte of output or maybe we made some obscure user error, as with using the name "write." We don't plan to find out.

Disappointed Again

We don't care because the experiment had already failed before that last problem turned up. The C notation, despite its difficulties, is certainly usable, even elegant. It's great fun to eliminate lines of code by embedding increments, decrements, and assignments in the arguments of functions. But consider: even if the example program wrote to disk as fast as it reads, it would still be

- a 16KB program from a 2KB source file
- with a throughput of 160 bytes/second

A hypothetical assembly program would be a 2KB object file with a throughput near 1000 bytes/second (if challenged, we'll write it to prove the point).

Consider also the quality of code generated by this, a typical one-pass compiler that is within a factor of 2 of the current state of the industry. The object file is full of things like

```
XCHG
INX H
INX H
XCHG
```

to increment the DE register, and

```
LXI H,0000
XCHG
```

to put zero in it. A simple peephole optimizer would doubtless bring Aztec C into competition with its faster rivals. But peephole optimization wouldn't fix the grosser blunders it shares with all one-pass programs, like recalculating a value that already exists in one of the registers, or calculating an expression on every pass of a for-loop even though the elements of the expression aren't changed in it. That requires global optimization.

Until somebody produces a genuine *optimizing* compiler for C (or Modula 2, or what-have-you), the software developer will be faced with a choice between writing quick, small, machine-dependent code in assembly language, or writing sluggish, bloated, portable code in C. It's a nasty dilemma, and we wish somebody would produce the compiler that would get us off its horns.

DDJ

(Listings begin on page 14)

Dr. Dobb's Clinic (Text begins on page 10)

Listing One

```
#include "stdio.h"
#define MAXIDS 500
#define MAXLEN 9
#define IGNORAD 0x0000

struct item {/* one symbol table entry */
     unsigned addr;
     char id[MAXLEN];
     };

int readsym(s)
struct item *s;
{ /* read one symbol table item from stdin */
     s->addr = IGNORAD;
     return ( scanf("%4x %8s",&(s->addr),s->id) );
}

int readall(sa)
struct item sa[];
{ /* read all symbol table items from stdin, count them, report full */
    int n;
    n = 0;
    while ( readsym(sa+n) != EOF ) {
        if ( (sa+n)->addr != IGNORAD ) {
            if ( (++n) == MAXIDS ) {
                fprintf(stderr,"too many items, %d will be used\n",MAXIDS);
                break;
            }
        }
    }
    return (n);
}

swap(p,q)
struct item *p, *q;
{/* exchange items p and q */
    struct item t;
    t = *p; *p = *q; *q = t;
}

quick(lo, hi, p)
struct item p[];
int lo,hi;
{/* sort an array of items using standard quicksort idiom */
    int i, j;
    unsigned pivot;
    if (lo < hi) {
        pivot = p[hi].addr;
        for (i = lo, j = hi ; i < j ; ) {
            while (i < j && p[i].addr <= pivot ) ++i;
            while (j > i && p[j].addr >= pivot ) --j;
            if (i < j) swap(&p[i],&p[j]);
```

```
        }
        swap(&p[i],&p[hi]);
        quick(lo,i-1,p);
        quick(i+1,hi,p);
    }
}

xprint(n)
unsigned n;
{/* display hex address the hard way */
    static char digits[16] = { '0','1','2','3','4','5','6','7',
                               '8','9','A','B','C','D','E','F' };
    int c;
    for ( c = 12 ; c >= 0 ; c -= 4 )
        putc(digits[ (n >> c) & 0x000f ],stdout);
}

display(p,n)
struct item *p;
int n;
{/* write n symbols starting at p */
    for ( ; n>0 ; --n, ++p ) {
        xprint(p->addr);
        printf(" %s\n",p->id);
    }
}

main()
{
    struct item table[MAXIDS];
    int count;
fprintf(stderr,"reading\n");
    count = readall(table);
fprintf(stderr,"sorting\n");
    quick(0,count-1,table);
fprintf(stderr,"writing\n");
    display(table,count);
}
```

End Listing One

Listing Two

```
#c.initial
TYPE HISTOGRAM BOUNDS 100,4100
.INITIAL = C521
.COLLECT = C524
.DISPLAY = C527
#uffff,.collect
 --ME- A=BE B=0535 D=00CA H=A8BB S=A8B1 P=0593 CALL 0486 .DISPLAY_
*0518
#uffff,.collect
 ---E- A=5F B=048B D=A8AC H=A8EF S=A8A7 P=0518 ORA  A
*05D3
```

(Continued on page 18)

Listing Two

```
#c.display
HISTOGRAM:
ADDR      RELATIVE FREQUENCY, LARGEST VALUE = 9C03
....
0400 ******
0500 ************************************
0600 **********
0700 ************************************************************************
....
1E00 ***
1F00 **
2000 **********
2100 **
....
2A00 ******
....
3500 **********************
3600 ******************
3700 ****************
....
3900 ****
3A00 **
....
4100 *
```

End Listing Two

Listing Three

```
 c.initial
TYPE HISTOGRAM BOUNDS 400,800
.INITIAL = C521
.COLLECT = C524
.DISPLAY = C527
#uffff,.collect
 ----- A=01 B=36FA D=A861 H=A861 S=A861 P=05D3 SPHL
*36F4
#c.display
HISTOGRAM:
ADDR      RELATIVE FREQUENCY, LARGEST VALUE = 1DCA
....
0410
0420 *
0430 **
0440 *
0450 **
0460 *
0470 **
0480
0490
04A0
04B0
04C0
04D0
04E0
```

```
04F0 **
0500 **
0510 **
0520
....
05B0 **********
05C0 **************************
05D0 ************************************
05E0 *****
05F0 *******
0600 *******
0610 ***
0620 **
0630 ********
0640 *****
....
0700 ***************
0710 ***********
....
0760 *****
0770 ***********
0780 ********************************************
0790 *
07A0 **********
07B0 ************************************************************************
07C0 ***
....
0800 *
```

End Listing Three

A Kernal for the MC 68000

by Steve Passe

Sixteen bits and nowhere to go! Being a "hardware" type I naturally had to have a 16-bitter as soon as I could manage it. After letting the dust settle a bit I decided to go with the 68000 processor from Motorola. The board I chose (the M68K from Educational Microcomputer Systems) has Motorola's Macsbug monitor in PROM, but other than that I had no software to run my new friend.

Several months later, after punching out a cross-assembler in C for software development, I was ready to start writing code. But what? I have always been fascinated by real-time software, and multiple tasking is supposed to be one of the things that the 68000 is designed for. So, enter the kernal.

The Kernal

A kernal is a software device that distributes CPU time among concurrently running processes. A process is an individual task, or program, that runs asynchronously with other tasks. Since the kernal is continuously switching between tasks at a high rate of speed, it appears to a human observer that all tasks are running simultaneously. Responding to both hardware interrupts and software traps, the kernal decides which process gets the next use of the CPU, how long it may use it, and in what order other processes must wait.

System resources other than the CPU may also be shared by creating device manager processes for each port, disk drive, and so on. The process controls the device, carrying out requests for device usage from other processes. It also may return either data or status information to the requesting process.

As an example, a device manager would be written to control each terminal port on a system. A user program wishing to do I/O would generate a software trap to the kernal, passing a request for either input or output. The data would typically be placed into a common buffer before an output request, or taken from the buffer after an input request.

Upon receiving such a request from a process, the kernal would stop it and save all necessary registers in memory. It would then start the device managing task and pass the request to it. This task would initiate the I/O and give control back to the kernal. The kernal would then run some arbitrary task until receiving an interrupt from the hardware signalling that the I/O is complete. It then awakens the manager, which passes control back to the requesting process along with a result message of the I/O process.

Although this seems quite complicated on the surface, it allows more efficient use of the CPU since it is not wasting time while waiting on the relatively slow speed of port hardware. It also creates a level of system protection since the device manager may refuse to grant requests based on permission algorithms, reservations, etc.

The 68000

The Motorola 68000 offers all the hardware features necessary for efficient implementation of such a kernal. It allows a total of 192 user-generated interrupts, either software or hardware generated. Each such interrupt has a corresponding four-byte vector in the first page of memory that holds a jump address to a service routine. It also has two different operating modes: supervisor and user. When in user mode, certain instructions may not be used, allowing direct hardware protection of sensitive instructions. Furthermore, if utilized, the hardware can address memory so that no process running in user mode can even access memory set aside for the supervisor state/code.

The assembler syntax used in the listing (page 22) is fairly straightforward, consisting of mnemonics and pseudo-ops as defined by Motorola. Comments are preceded by an asterisk (*) on comment-only lines, or may follow the operand fields of statements without the asterisk. The ".b", ".w", and ".l" extensions of mnemonics specify byte, word, and long word data sizes. The pound sign (#) marks an immediate value.

The Code

The code as presented is far from a complete system. It is a workable starting point, demonstrating the features of a multi-tasking kernal. First I will explain what it does as written, then I'll go into various ways it might be modified for particular uses.

The first set of equates defines the memory/port map. It should be noted that the 68000 uses memory-mapped I/O, thus allowing any instruction that manipulates memory to also access a port. The next section, commented out, describes the exception (hardware and software interrupt) vector table utilized by the 68000. When implementing this kernal, you would set whichever of these vectors the application required.

Beyond this is the beginning of the kernal data area (800hex). A maximum of eight concurrent processes are allowed, with three being set up at the start. Following this is a process descriptor definition. It contains a pointer to the next process descriptor, a priority flag, and space to store the entire register set of the CPU. The final lines set aside additional descriptor space for each of the possible tasks.

A series of pointers is then allocated. *Running* is a pointer to the process descriptor of the currently running task. *Ready* is a pointer to the head of the queue of tasks waiting for CPU time. *Dead* is a pointer to a queue of unused process descriptors. *Conditions* is an array of pointers, one for each process, in which the process descriptor's address is stored when it is "sleeping," i.e., when it is waiting to be restarted by another task. Finally, *device 1* and *device 2* point to queues of tasks waiting on those devices.

The first part of the kernal is its initialization entry point. It sets up its stack, initializes the needed exception vectors, and queues the processes into the running and ready slots. It then goes on to initialize three very simple tasks for demonstration purposes. This includes setting a program counter, stack pointer, and status register value for each. Remaining unused descriptors are put into the *dead* queue for future use. Finally, the stack pointer and status register values for the first task are pushed onto the stack. It is then started with an *rte* instruction. This is the standard method of causing the 68000 to return to a task that generated an exception. The task will then run until an exception (interrupt or trap) occurs.

Steve Passe, New Ideas, P.O. Box 629, Englewood, Colorado 80151.

Copyright © 1983 Steve Passe. All rights reserved.

Permission is granted for non-commercial use and distribution. Commercial use without the author's written permission is forbidden.

The Kernal Body

The body of the kernal has three entry points. The major entry point is *slice*. It is addressed by the exception vector reserved for hardware clock interrupts. In my case this amounts to a button on the front panel for clock simulation purposes. The *wait* entry point is called (with a trap #10) by a task wishing to sleep while awaiting some occurrence within the system. The condition (i.e., the condition slot) it wishes to wait on is passed in data register 7. The corresponding *signal* entry point (trap #11) is used by another task to cause awakening of a task. The particular condition it wishes to signal is passed in data register 7. The *slice_trap* entry (trap #12) is used by a process to give up the CPU to the next waiting process.

Other routines should be accessed only from within the kernal. *Store_runner* and *restore_runner* save and restore the machine state when switching tasks. *Switch* is used by the kernal to place the running task into the ready queue and retrieve a waiting task.

The Tasks

The tasks I have written so far do nothing more than give a visual display of their existence. Task one prints a series of ASCII one's (1) then waits on condition 1 to become true. Task two prints a series of two's (2) then waits on condition 2. Task three prints a series of three's (3), signals task two, then signals task one. This whole process continues until a "clock" interrupt occurs. This clocking causes the current task to suspend and the next task to execute. The device managers for the ports described above have not yet been written.

Modifying For The Real World

Obviously the first changes to make would be to remove the fake tasks and write useful replacements. A real system clock would be added to cause time slicing. The address of the slice routine in the kernal would be placed in the hardware interrupt vector slot used by the clock.

If the kernal were to be used for real-time control of specific hardware devices, individual tasks would be created that manage each device. This would include handshaking via interrupts and assignment of relative priorities with the priority word in each descriptor.

If used as the basis of an operating system, a user interface task would be needed, as well as device managers for each port. Processes for a file system and a media storage manager would be needed. The operating system written in Small-C, published in the March '83 *DDJ*, would be an interesting starting point.

It would be possible to save task switching time by saving only those registers known to be used by user processes.

The process descriptor allocation could be made completely dynamic if it were to be kept just above each task's stack pointer. This scheme would only work with simpler hardware that didn't map memory according to the current operating mode, since the descriptor normally accessed from supervisor mode would now reside in user memory.

This software is free of restrictions as to non-commercial use and distribution. Commercial use without author's permission is forbidden. I will try to keep the code on-line on my dial-up system around the time it is published. I also intend to keep various public domain 68000 utilities available for callers. The system is accessible 24 hours, seven days a week, at (303) 781-4937. Public domain code, ideas, and discussion about the 68000 are all welcome.

DDJ

(Listing begins below)

MC68000 Kernal Listing (Text begins on page 20)

```
*****************************************************************************
*                                                                           *
*                  A Kernal for the MC 68000 processor                      *
*                                                                           *
*        (C) Copyright 1983 Steve Passe,  all rights reserved               *
*                                                                           *
*****************************************************************************

0009 000000                                 first_page:
             0000 0000                       equ     $0              exception vec
                                             tors
0010 000000                                 kernal:
             0000 0800                       equ     $800            init entry to
                                              kernal
0011 000000                                 kernal_data:
             0000 7000                       equ     $7000           user scratch
                                             memory on ems

0013 000000                                 user_sr_mask:
             0000 0000                       equ     0               no trace/user
                                             /lowest int.

0015 000000                                 auto_v_int:
             0000 0064                       equ     $64             address of au
                                             to vetor #1
0016 000000                                 trap_0:
             0000 0080                       equ     $80             address of tr
```

MC68000 Kernal Listing (Listing continued, text begins on page 20)

```
                                                    ap #0 vector

 0018 000000                                port:
              0003 FF01                      equ     $03ff01                  acia
                                                    #1 status on ems
 0019 000000                                int_1_ack:
              0003 FF30                      equ     $03ff30                  level
                                                     1 (clock) int ack
*
*       map of exception vectors, not used when running under macsbug
*
*                       org     first_page+8    first 8 cast in silicon
*
*               dc.l    error_vector            2: bus error
*               dc.l    error_vector            3: address error
*               dc.l    error_vector            4: illegal instruction
*               dc.l    error_vector            5: divide by zero
*               dc.l    error_vector            6: out of bounds
*               dc.l    error_vector            7: overflow
*               dc.l    error_vector            8: privilege violation
*               dc.l    error_vector            9: trace routine
*               dc.l    error_vector            10: 1010 psuedo code
*               dc.l    error_vector            11: 1111 pseudo code
*
*       vectors 12 thru 23 reserved by Motorola, to be
*       really safe I suppose you should init them...
*
*                       org     first_page+$5c  address of vector #24
*
*               dc.l    error_vector            24: spurious interruot
*               dc.l    error_vector            25: #1 autovector, system clock
*               dc.l    error_vector            26: level 2 int. autovector
*               dc.l    error_vector            27: level 3 int. autovector
*               dc.l    error_vector            28: level 4 int. autovector
*               dc.l    error_vector            29: #5 autovector, serial oorts
*               dc.l    error_vector            30: #6 autovector, parallel o.
*               dc.l    error_vector            31: #7 autovector, abort switch
*               dc.l    error_vector            32: trap #0 vector
*               dc.l    error_vector            33: trap #1 vector
*               dc.l    error_vector            34: trap #2 vector
*               dc.l    error_vector            35: trap #3 vector
*               dc.l    error_vector            36: trap #4 vector
*               dc.l    error_vector            37: trap #5 vector
*               dc.l    error_vector            38: trap #6 vector
*               dc.l    error_vector            39: trap #7 vector
*               dc.l    error_vector            40: trap #8 vector
*               dc.l    error_vector            41: trap #9 vector
*               dc.l    wait                    42: #10, kernal wait entry
*               dc.l    signal                  43: #11, kernal signal entry
*               dc.l    slice_trap              44: #12, kernal slice entry
*               dc.l    error_vector            45: trap #13 vector
*               dc.l    error_vector            46: trap #14 vector
*
*       comment out trap #15 init to retain macsbug console i/o trap...
*               dc.l    error_vector            47: trap #15 vector
*
*
*       vectors 48 thru 63 reserved by Motorola, to be
```

```
*       really safe I suppose you should init them also...
*
*       gonna cheat here, 64-255 are for hardware interrupts that
*       generate their own vector number.  If your hardware can
*       generate such interrupts these must also be initialized.
*


 0080 007000 7000                          org     kernal_data
*
*       maximum # of processes in the kernal
*

 0085 007000                           processes:
              0000 0008                   equ     8               # of processe
                                                  s supported
 0086 007000                           runners:
              0000 0003                   equ     3               # of running
                                                  processes

*
*       the process descripters, need one for each possible active process
*

 0092 007000                           process_1:
 0093 007000                           next_process:
              0000 0000                   equ     *-process_1     offset to nex
                                                  t process
 0094 007000 0000 7214                    dc.l    process_x       points to nex
                                                  t process in queue
 0095 007004                           priority:
              0000 0004                   equ     *-process_1     offset to pro
                                                  cess priority
 0096 007004 0063                         dc.w    99              priority of t
                                                  his process
 0097 007006                           sr_slot:
              0000 0006                   equ     *-process_1     offset to sta
                                                  tus reg.
 0098 007006 0000                         dc.w    0               value of stat
                                                  us register
 0099 007008                           pc_slot:
              0000 0008                   equ     *-process_1     offset to pro
                                                  g. cntr.
 0100 007008 0000 0D6C                    dc.l    dummy           value of prog
                                                  ram counter
 0101 00700C                           d0_slot:
              0000 000C                   equ     *-process_1     offset to d0
                                                  copy
 0102 00700C 0000 0000                    dc.l    0               value of d0 f
                                                  or this process
 0103 007010 0000 0000                    dc.l    0               value of d1
 0104 007014 0000 0000                    dc.l    0               value of d2
 0105 007018 0000 0000                    dc.l    0               value of d3
```

(Continued on next page)

MC68000 Kernal Listing (Listing continued, text begins on page 20)

```
0106 00701C 0000 0000                dc.l    0               value of d4
0107 007020 0000 0000                dc.l    0               value of d5
0108 007024 0000 0000                dc.l    0               value of d6
0109 007028 0000 0000                dc.l    0               value of d7
0110 00702C 0000 0000                dc.l    0               value of a0 f
                                     or this process
0111 007030 0000 0000                dc.l    0               value of a1
0112 007034 0000 0000                dc.l    0               value of a2
0113 007038 0000 0000                dc.l    0               value of a3
0114 00703C 0000 0000                dc.l    0               value of a4
0115 007040 0000 0000                dc.l    0               value of a5
0116 007044                         a6_slot:
             0000 0044               equ     *-process_1     offset to a6
                                     copy
0117 007044 0000 0000                dc.l    0               value of a6
0118 007048                         a7_slot:
             0000 0048               equ     *-process_1     offset to a7
                                     copy
0119 007048 0000 0E7C                dc.l    dummy_so        value of a7
0120 00704C 0000 0000                ds      0               insure alignm
                                     ent
0121 00704C                         process_size:
             0000 004C               equ     *-process_1     size of a pro
                                     cess descripter

0123 00704C                         process_2:
             0000 01C8               ds.b    (processes-2)*process_size
                                             less #1 & dummy

0125 007214                         process_x:
             0000 004C               ds.b    process_size    dummy process

0127 007260                         process_space:
             0000 0260               equ     *-process_1

*       the queue pointers...
*

0133 007260                         running:
             0000 0000               dc.l    0                       point
                                     er to running process

0135 007264                         ready:
             0000 0000               dc.l    0                       point
                                     er to head of ready queue

0137 007268                         dead:
             0000 0000               dc.l    0                       point
                                     er to head of dead queue

*
*       conditions (args to wait and signal), only
*       one process can wait for any one condition
*       in this version...
*
```

```
 0145 00726C                                conditions:
             0000 0008                       ds.l    processes               * con
                                                     dition slots, one for
                                                    * each possible process

*
*       device 1 is acia #1 for this machine
*
 0152 00728C                                device_1:
             0000 0001                       ds.l    1                       ptr t
                                                     o head of device_1 queue

*
*       device 2 is acia #2
*
 0158 007290                                device_2:
             0000 0001                       ds.l    1                       ptr t
                                                     o head of device_2 queue

*************************************************************************
*                                                                       *
*       The initial entry point to the kernal....                       *
*                                                                       *
*                                                                       *
*                                                                       *
*************************************************************************

 0168 000800 0800                           org     kernal

 0170 000800                                kernal_init:

         *
         *       setup the machine
         *
 0175 000800 46FC 2700                       move.w  #$2700,sr
                                                     no int, su
 0176 000804 2E7C 0000 7C00                  move.l  #$7c00,sp
                                                     kernal stack

         *
         *       set the needed vectors
         *
 0181 00080A 21FC 0000 09A2 0064             move.l  #slice,auto_v_int
                                                     hardware clock, v1
 0182 000812 21FC 0000 0E86 0074             move.l  #serial_port,auto_v_int+(4*4
                                                    )        acia interrupts, v5
 0183 00081A 21FC 0000 0E88 0078             move.l  #parallel_port,auto_v_int+(5
                                                    *4)      parallel port int, v6

 0184 000822 21FC 0000 0E8A 007C             move.l  #abort_button,auto_v_int+(6*
                                                    4)       abort switch, v7
** this is temp till a system clock is installed:
 0186 00082A 21FC 0000 09A2 007C             move.l  #slice,auto_v_int+(6*4)
```

(Continued on next page)

MC68000 Kernal Listing (Listing continued, text begins on page 20)

```
**
 0188 000832 21FC 0000 0958 00A8          move.l  #wait,trap_0+(10*4)     kerna
                                                  l wait entry, trap #10
 0189 00083A 21FC 0000 0976 00AC          move.l  #signal,trap_0+(11*4)   kerna
                                                  l signal entry, #11
 0190 000842 21FC 0000 09B8 00B0          move.l  #slice_trap,trap_0+(12*4) (s
                                                  oftware) process slice, #12

        *
        * initialize and queue all the processes...
        *
 0195 00084A 4DF8 7000                    lea     process_1,a6
                                                          a6 holds ptr to runni
                                                  ng
 0196 00084E 21CE 7260                    move.l  a6,running
                                                          running points to p_1

 0197 000852 2A4E                         move.l  a6,a5
                                                          a5 points @ process_1

 0198 000854 DBFC 0000 00E4               add.l   #(runners*process_size),a5
                                                          x runners
 0199 00085A 21CD 7268                    move.l  a5,dead
                                                                  dead points @
                                                   process_?
**       move.l          #process_x,ready                 ready points @ dummy
 0201 00085E 21FC 0000 704C 7264          move.l  #process_2,ready
                                                          ready points @ p_2

       *
       * init the dead for future
       *
0206 000866 3E3C 0213                    move.w  #((processes-1)*process_size
                                                  )-1,d7   bytes, less dbf
0207 00086A 284E                         move.l  a6,a4
                                                          a4 points at base
0208 00086C 4BF8 704C                    lea     process_2,a5
                                                          a5 points @ process_2

0209 000870                             next_byte:
0210 000870 1ADC                         move.b  (a4)+,(a5)+
                                                          stuff a byte
0211 000872 51CF FFFC                    dbf     d7,next_byte
                                                          another byte

       *
       * init first and last process reg values
       *
0216 000876 2D7C 0000 09FC 0008          move.l  #task_1,pc_slot(a6)
                                                          setup program counter

0217 00087E 3D7C 0001 0004               move.w  #1,priority(a6)
                                                                  setup priorit
                                                  y
```

```
0218 000884 3D7C 0000 0006                move.w  #user_sr_mask,sr_slot(a6)
                                                  setup status register
0219 00088A 2D7C 0000 0B20 0048           move.l  #task_1_sp,a7_slot(a6)
                                                  setup stack

0221 000892 4BF8 704C                     lea     process_2,a5
                                                  a5 holds desrc. base
0222 000896 2ABC 0000 7098                move.l  #process_2+process_size,(a5)
                                                  point to next process

0223 00089C 2B7C 0000 0B20 0008           move.l  #task_2,pc_slot(a5)
                                                  setup program counter

0224 0008A4 3B7C 0001 0004                move.w  #1,priority(a5)
                                                        setup priorit
                                                  y
0225 0008AA 3B7C 0000 0006                move.w  #user_sr_mask,sr_slot(a5)
                                                  setup status register

0226 0008B0 2B7C 0000 0C44 0048           move.l  #task_2_sp,a7_slot(a5)
                                                  setup stack

0228 0008B8 DBFC 0000 004C                add.l   #process_size,a5
                                                  a5 holds desrc. base
0229 0008BE 2B7C 0000 0C44 0008           move.l  #task_3,pc_slot(a5)
                                                  setup program counter

0230 0008C6 3B7C 0001 0004                move.w  #1,priority(a5)
                                                        setup priorit
                                                  y
0231 0008CC 3B7C 0000 0006                move.w  #user_sr_mask,sr_slot(a5)
                                                  setup status register

0232 0008D2 2B7C 0000 0D6C 0048           move.l  #task_3_sp,a7_slot(a5)
                                                  setup stack

0234 0008DA 4BF8 7214                     lea     process_x,a5
                                                  a5 holds dummy proces
                                                  s
0235 0008DE 2B7C 0000 0D6C 0008           move.l  #dummy,pc_slot(a5)
                                                  setup program counter

0236 0008E6 3B7C 0000 0006                move.w  #user_sr_mask,sr_slot(a5)
                                                  setup status register
0237 0008EC 2B7C 0000 0E7C 0048           move.l  #dummy_sp,a7_slot(a5)
                                                  setup stack

        *
        * link rest of processes into dead
        *
0242 0008F4 2A78 7268                     move.l  dead,a5
                                                        dead pointer
```

(Continued on next page)

MC68000 Kernal Listing (Listing continued, text begins on page 20)

```
0243 0008F8 7E03                        moveq   #processes-(runners+1+1),d7
                                                minus runners/dummy/d
                                        bf
0244 0008FA                         next_dead:
0245 0008FA DBFC 0000 004C              add.l   #process_size,a5
                                                point to next process

0246 000900 2B4D FFB4                   move.l  a5,-process_size(a5)
                                                into last prcs.next
0247 000904 51CF FFF4                   dbf     d7,next_dead
                                                continue

       *
       * go for it!
       *
0252 000908 207C 0000 0B20              move.l  #task_1_sp,a0
                                                task 1 sp into a0...
0253 00090E 4E60                        move.l  a0,usp
                                                ...then into user sp
0254 000910 2F3C 0000 09FC              move.l  #task_1,-(sp)
                                                setup rte, task 1 add
                                        r.
0255 000916 3F3C 0000                   move.w  #0,-(sp)
                                                 and status reg. valu
                                        e
0256 00091A 4E73                        rte
```

```
*************************************************************************
*                                                                       *
*       the body of the kernal code...                                  *
*                                                                       *
*       reg d7 is used to pass condition #                              *
*       reg a6 is reserved by kernal to keep pointer to running         *
*                                                                       *
*       entry points:                                                   *
*                       1: wait,        trap #10                        *
*                       2: signal,      trap #11                        *
*                       3: slice_trap,  trap #12                        *
*                                                                       *
*************************************************************************
```

```
0272 00091C                         kernal_body:

0274 00091C                         store_runner:
                                        *
                *       save the running process in it's descripter
                *
0277 00091C 2F0E                        move.l  a6,-(sp)
                                                push a6 onto stack
0278 00091E 2C78 7260                   move.l  running,a6
                                                current runner into a
                                        6
0279 000922 48EE 3FFF 000C              movem.l d0-d7/a0-a5,d0_slot(a6)
                                                        store the dat
                                        a & addr. regs.
```

```
0280 000928 2D5F 0044                    move.l  (sp)+,a6_slot(a6)
                                                 poo saved a6
0281 00092C 3D6F 0004 0006               move.w  4(sp),sr_slot(a6)
                                                 store status register
0282 000932 2D6F 0006 0008               move.l  6(sp),pc_slot(a6)
                                                 store the program cou
                                                 nter
0283 000938 2F57 0006                    move.l  (sp),6(sp)
                                                 put return over sr/pc
0284 00093C 5C8F                         addq.l  #6,sp
                                                 setup return
0285 00093E 082E 0005 0006               btst    #5,sr_slot(a6)
                                                 were we supervisor be
                                                 fore?
0286 000944 6608                         bne.s   is_super
                                                 yes, don't need usp..
                                                 .
0287 000946 4E6C                         move.l  usp,a4
                                                 no, get users stack p
                                                 ointer
0288 000948 2D4C 0048                    move.l  a4,a7_slot(a6)
                                                 and place in descript
                                                 er
0289 00094C 4E75                         rts
0290 00094E                              is_super:
0291 00094E 2D4F 0048                    move.l  sp,a7_slot(a6)
                                                 present ssp value
0292 000952 58AE 0048                    addq.l  #4,a7_slot(a6)
                                                 value before this sub
                                                 routine
0293 000956 4E75                         rts

0295 000958                              wait:
0296 000958 007C 0700                    or      #$0700,sr
                                                 block interruots
0297 00095C 61BE                         bsr.s   store_runner
                                                 save this process
0298 00095E CEFC 0004                    mulu    #4,d7
                                                 # by pointer size
0299 000962 4BF8 726C                    lea     conditions,a5
                                                 base of conditions
0300 000966 2B8E 7000                    move.l  a6,0(a5,d7)
                                                 store process in cond
                                                 ition x
0301 00096A                              run:
0302 00096A 2C78 7264                    move.l  ready,a6
                                                 get ready from queue
0303 00096E 21D6 7264                    move.l  (a6),ready
                                                 new head of ready que
                                                 ue
0304 000972 614A                         bsr.s   restore_runner
                                                 setup next process
0305 000974 4E73                         rte
```

(Continued on next page)

MC68000 Kernal Listing (Listing continued, text begins on page 20)

```
0307 000976                          signal:
0308 000976 007C 0700                 or       #$0700,sr
                                                block interrupts
0309 00097A CEFC 0004                 mulu     #4,d7
                                                make offset to condit
                                      ion
0310 00097E 0687 0000 726C            add.l    #conditions,d7
                                                add base of condition
                                      s
0311 000984 CF8E                      exg      d7,a6
                                                setup indirection on
                                      a6
0312 000986 4A96                      tst.l    (a6)
                                                test for null
0313 000988 CF8E                      exg      d7,a6
                                                restore prior order
0314 00098A 6602                      bne.s    switch
                                                if waiting go for it
0315 00098C 4E73                      rte

0317 00098E                          switch:
0318 00098E 618C                      bsr.s    store_runner
                                                save runner
0319 000990 2F07                      move.l   d7,-(sp)
0320 000992 49F8 7264                 lea      ready,a4
                                                start at head of read
                                      y queue
0321 000996 6150                      bsr.s    link_queue
                                                put into ready queue
0322 000998 2A5F                      move.l   (sp)+,a5
                                                setup indirection aga
                                      in
0323 00099A 2C55                      move.l   (a5),a6
                                                      get the waite
                                      r
0324 00099C 4295                      clr.l    (a5)
                                                not waiting anymore
0325 00099E 611E                      bsr.s    restore_runner
                                                restore new runner
0326 0009A0 4E73                      rte

0328 0009A2                          slice:
0329 0009A2 007C 0700                 or       #$0700,sr
                                                block interrupts
0330 0009A6 4A39 0003 FF30            tst.b    int_1_ack
                                                clear interrupt
0331 0009AC                          soft_slice:
0332 0009AC 6100 FF6E                 bsr      store_runner
                                                save runners register
                                      s
0333 0009B0 49F8 7264                 lea      ready,a4
                                                start at head of read
                                      y queue
0334 0009B4 6132                      bsr.s    link_queue
                                                link runner into read
```

```
                                                        y queue
0335 0009B6 60B2                                bra.s   run
                                                        go for ready

0337 0009B8                                    slice_trap:
                                                        * entry to slice from softwa
                                                        re
0338 0009B8 007C 0700                           or      #$0700,sr
                                                        block interruots
0339 0009BC 60EE                                bra.s   soft_slice

0341 0009BE                                    restore_runner:
                                                        *
                *       restore runner from descripter (in a6 when called)
                *

0345 0009BE 21CE 7260                           move.l  a6,running
                                                        record new runner
0346 0009C2 265F                                move.l  (sp)+,a3
                                                        save return, clear st
                                                        ack
0347 0009C4 286E 0048                           move.l  a7_slot(a6),a4
                                                        stack to restore
0348 0009C8 082E 0005 0006                      btst    #5,sr_slot(a6)
                                                        were we supervisor be
                                                        fore?
0349 0009CE 6604                                bne.s   was_super
                                                        yes, don't need usp
0350 0009D0 4E64                                move.l  a4,usp
                                                        ...then into usp
0351 0009D2 6002                                bra.s   sp_set
                                                        we hope!
0352 0009D4                                    was_super:
0353 0009D4 2E4C                                move.l  a4,sp
                                                        ssp = old ssp
0354 0009D6                                    sp_set:
0355 0009D6 2F2E 0008                           move.l  pc_slot(a6),-(sp)
                                                        restore the program c
                                                        ounter
0356 0009DA 3F2E 0006                           move.w  sr_slot(a6),-(sp)
                                                        restore status regist
                                                        er
0357 0009DE 2F0B                                move.l  a3,-(sp)
                                                        restore return
0358 0009E0 4CEE 7FFF 000C                      movem.l d0_slot(a6),d0-d7/a0-a6
                                                        restore data & addr.
                                                        regs.
0359 0009E6 4E75                                rts

0361 0009E8                                    link_queue:
                                                        *
                *       link a process into the queue pointer to by a4
                *
0364 0009E8 264C                                move.l  a4,a3
                                                        remember last
```

(Continued on page 38)

MC68000 Kernal Listing (Listing continued, text begins on page 20)

```
0365 0009EA 2854                        move.l  (a4),a4
                                                        next link int
                                        o head
0366 0009EC 3E2C 0004                   move.w  priority(a4),d7
                                                        get priority
                                        into dat reg.
0367 0009F0 BE6E 0004                   cmp.w   priority(a6),d7
                                                        compare runni
                                        ng and head
0368 0009F4 63F2                        bls.s   link_queue
                                                keep trying
0369 0009F6 268E                        move.l  a6,(a3)
                                                        last.next ==
                                        running
0370 0009F8 2C8C                        move.l  a4,(a6)
                                                        (former) runn
                                        ing.next == link
0371 0009FA 4E75                        rts
*
*       the first process
*
 0377 0009FC                            task_1:
 0378 0009FC 223C 0000 0050              move.l  #80,d1
 0379 000A02                            t_1:
              207C 0003 FF01             move.l  #port,a0                acia
                                        #1
 0380 000A08 103C 0031                   move.b  #'1',d0
                                                why?
 0381 000A0C 4EB8 0E8E                   jsr     put_char                send
                                        it
 0382 000A10 7001                        moveq   #1,d0                   one t
                                        enth sec
 0383 000A12 4EB8 0E7C                   jsr     sleep_tenth             kill
                                        time
 0384 000A16 51C9 FFEA                   dbf     d1,t_1
 0385 000A1A 7E01                        moveq   #1,d7                   wait
                                        on condition one
 0386 000A1C 4E4A                        trap    #10
 0387 000A1E 60DC                        bra.s   task_1                  forev
                                        er
 0388 000A20 0000 0080                   ds      128                     enoug
                                        h?
 0389 000B20 0000 0000                   ds      0                       insur
                                        e alignment
 0390 000B20                            task_1_sp:
*
*       the second process
*
 0396 000B20                            task_2:
 0397 000B20 223C 0000 0050              move.l  #80,d1
 0398 000B26                            t_2:
              207C 0003 FF01             move.l  #port,a0                acia
                                        #1
 0399 000B2C 103C 0032                   move.b  #'2',d0
                                                why not!
 0400 000B30 4EB8 0E8E                   jsr     put_char                send
                                        it
```

```
0401 000B34 7001                        moveq   #1,d0
0402 000B36 4EB8 0E7C                   jsr     sleep_tenth             kill
                                                time
0403 000B3A 51C9 FFEA                   dbf     d1,t_2
0404 000B3E 7E02                        moveq   #2,d7                   wait
                                                on condition two
0405 000B40 4E4A                        trap    #10
0406 000B42 60DC                        bra.s   task_2                  forev
                                                er
0407 000B44 0000 0080                   ds      128                     i hop
                                                e so
0408 000C44 0000 0000                   ds      0                       insur
                                                e alignment
0409 000C44                             task_2_sp:
*
*       the third process
*
0415 000C44                             task_3:
0416 000C44 223C 0000 0050              move.l  #80,d1
0417 000C4A                             t_3:
             207C 0003 FF01             move.l  #port,a0                acia
                                                #1
0418 000C50 103C 0033                   move.b  #'3',d0
0419 000C54 4EB8 0E8E                   jsr     put_char                send
                                                it
0420 000C58 7001                        moveq   #1,d0
0421 000C5A 4EB8 0E7C                   jsr     sleep_tenth             kill
                                                time
0422 000C5E 51C9 FFEA                   dbf     d1,t_3
0423 000C62 7E02                        moveq   #2,d7                   signa
                                                l two
0424 000C64 4E4B                        trap    #11
0425 000C66 7E01                        moveq   #1,d7                   signa
                                                l one
0426 000C68 4E4B                        trap    #11
0427 000C6A 60D8                        bra.s   task_3                  forev
                                                er
0428 000C6C 0000 0080                   ds      128
0429 000D6C 0000 0000                   ds      0                       insur
                                                e alignment
0430 000D6C                             task_3_sp:
*
*       the dummy process
*
0436 000D6C                             dummy:
0437 000D6C 103C 0021                   move.b  #'!',d0
                                                        why for?
0438 000D70 207C 0003 FF01              move.l  #port,a0                acia
                                                #1
0439 000D76 4EB8 0E8E                   jsr     put_char                send
                                                it
0440 000D7A 60F0                        bra.s   dummy                   forev
                                                er
0441 000D7C 0000 0080                   ds      128
0442 000E7C 0000 0000                   ds      0                       insur
                                                e alignment
0443 000E7C                             dummy_sp:
*
* sleep one tenth second
*
```

(Continued on next page)

MC68000 Kernal Listing (Listing continued, text begins on page 20)

```
0449 000E7C                                sleep_tenth:
0450 000E7C                                tenth_constant:
             0000 C350                      equ     50000
0451 000E7C C0FC C350                       mulu    #tenth_constant,d0      multi
                                                    ply by constant
0452 000E80                                s_t:
             51C8 FFFE                      dbf     d0,s_t
0453 000E84 4E75                            rts


0456 000E86                                serial_port:
                                                    *
                *       process serial port interrupts
                *
0459 000E86 4E73                            rte

0461 000E88                                parallel_port:
                                                    *
                *       process parallel port interrupts
                *
0464 000E88 4E73                            rte

0466 000E8A                                abort_button:
                                                    *
                *       process the abort button
                *
0469 000E8A 4E73                            rte

0471 000E8C                                error_vector:
                                                    *
                *       process misc. vectors called incorrectly...
                *
*
*       code to report error
*
0477 000E8C 4E73                            rte
*
*       routine to output a char to port addressed by register a0

*
*               a0:     address of port
*               d0:     byte to output
*

0487 000E8E                                put_char:
0488 000E8E 2F01                            move.l  d1,-(sp)
                                                    save d1
0489 000E90                                p_c:
0490 000E90 1210                            move.b  (a0),d1
                                                            load status w
                                                    ord
0491 000E92 0201 0002                       and.b   #2,d1
                                                    check tbuffer empty
0492 000E96 67F8                            beq.s   p_c
                                                    not empty yet...
0493 000E98 1140 0002                       move.b  d0,2(a0)
                                                    ready, ship it out
0494 000E9C 221F                            move.l  (sp)+,d1
                                                    restore d1
```

```
 0495 000E9E 4E75                    rts

 0497 000EA0                         end     k

symbol -                        hex value -    decimal -    atrb. -

 a6_slot                              44            68       0x03
 a7_slot                              48            72       0x03
 abort_button                        E8A          3722       0x01
 auto_v_int                           64           100       0x01
 conditions                         726C         29292       0x01
 d0_slot                               C            12       0x03
 dead                               7268         29288       0x01
 device_1                           728C         29324       0x01
 device_2                           7290         29328       0x01
 dummy                               D6C          3436       0x01
 dummy_sp                            E7C          3708       0x01
 error_vector                        E8C          3724       0x01
 first_page                            0             0       0x01
 int_1_ack                         3FF30        261936       0x01
 is_super                            94E          2382       0x01
 kernal                              800          2048       0x01
 kernal_body                         91C          2332       0x01
 kernal_data                        7000         28672       0x01
 kernal_init                         800          2048       0x01
 link_queue                          9E8          2536       0x01
 next_byte                           870          2160       0x01
 next_dead                           8FA          2298       0x01
 next_process                          0             0       0x03
 o_c                                 E90          3728       0x01
 parallel_port                       E88          3720       0x01
 pc_slot                               8             8       0x03
 port                              3FF01        261889       0x01
 priority                              4             4       0x03
 process_1                          7000         28672       0x01
 process_2                          704C         28748       0x01
 process_size                          4C            76       0x03
 process_space                        260           608       0x03
 process_x                          7214         29204       0x01
 processes                             8             8       0x01
 put_char                            E8E          3726       0x01
 ready                              7264         29284       0x01
 restore_runner                      9BE          2494       0x01
 run                                 96A          2410       0x01
 runners                               3             3       0x01
 running                            7260         29280       0x01
 s_t                                 E80          3712       0x01
 serial_port                         E86          3718       0x01
 signal                              976          2422       0x01
 sleep_tenth                         E7C          3708       0x01
 slice                               9A2          2466       0x01
 slice_trap                          9B8          2488       0x01
 soft_slice                          9AC          2476       0x01
 sp_set                              9D6          2518       0x01
 sr_slot                               6             6       0x03
 store_runner                        91C          2332       0x01
 switch                              98E          2446       0x01
 t_1                                 A02          2562       0x01
 t_2                                 B26          2854       0x01
 t_3                                 C4A          3146       0x01
 task_1                              9FC          2556       0x01
```

(Continued on page 696)

A DML Parser

by David Harry

David J. Harry, Ph.D., 3311 Lobit Dr., Dickinson, TX 77539.

Copyright © 1983 David J. Harry, Ph.D. All rights reserved.

Permission is granted for non-commercial use and distribution. Commercial use without the author's written permission is forbidden.

How many times have you finished all but the user interface for that great new utility and decided it's just too complicated to develop a sophisticated command processor? Writing a command string decoder or similar utility from scratch can be complicated and may involve more effort than is justified for the purpose. Such applications fall into a broad class of problems relating to the parsing of a source language. The theory of parsing and translation is well developed, and several fine tools are available that automatically generate a parser from a description of the source language. Unfortunately, most of these utilities (e.g., the Unix-based YACC) are part of a compiler generating system requiring extensive support routines and are not appropriate for more modest, microcomputer-based applications.

DML, the system I describe in this article, is an attempt to provide a simple, yet powerful parsing system for microcomputer users. Although it is not a complete compiler generating system, you may use it to generate the entire first pass of a two-pass compiler for any well-formed computer language. More importantly, however, the parsing phase of the system is designed to be imbedded in any application program that requires sophisticated syntax recognition. Once the syntax of the input grammar has been specified in the DML metalanguage, a parsing table is automatically generated that describes the specified syntactic structures and directs output in a user-specified fashion. The parsing table thus produced is linked into the application program along with the DML runtime code and voila! instant parser.

Moreover, as an added interest to those who may consider exploiting DML as a compiler generator, the way that data structures and semantic operations are handled in DML makes it highly extensible. That is, the "actions" invoked by the parser as well as the data structures maintained may be altered at will, and yet the syntactic description of the source language as well as the parsing table it describes need never be recompiled. This gives the DML system a Forth-like extensibility.

The parsing system I describe is based upon a special purpose metalanguage, modestly called DML. Readers familiar with the basic theory of parsing will recognize it as syntactically related to Backus-Naur Form (BNF). In principle at least, if the syntactic constructs of a language can be described completely in DML, the programs provided will parse that language. In practice, certain limitations do exist that will be discussed later, but the class of languages that may be parsed by the DML system is sufficiently large to encompass any language normally encountered; this includes any currently used computer language.

The DML parsing system is composed of four principal parts: (1) the semantic and syntactic constructs of the DML language itself; (2) the data structures that are machine-executable representations of a syntactic description in DML; and (3) the programs that use such data structures to parse input in the source language. The fourth component is a compiler that produces machine representations from DML specifications. Figure 1 (page 50) illustrates the interrelationships between the various parts. I will discuss each in turn.

DML Language Specifications

Perhaps the best way to describe DML is to present a simple and familiar syntax represented in the DML metalanguage. Listing One (page 56) is a DML representation of the command string syntax for the CP/M utility PIP. It demonstrates all of the syntactic forms comprising the DML metalanguage. Later in the article, I'll describe how to implement the accompanying programs, which parse a "pip" command string according to the specifications in Listing One.

A DML syntax specification is composed of a sequence of *productions*. Each production is composed of a left part, or *entry name*, followed by a ":=", followed by one or more syntactic constructs, and terminating with a ";". Disregard for a moment the symbols ":", "[", "]", "!", and all parenthesized constructs. Two principal classes of constructs remain: names enclosed in angle brackets, called *nonterminals*, and single symbols, possibly enclosed in double or single quotes, called *terminals*. Adjacent symbols are related as in BNF by an implicit boolean "and" operator (specified here as &). Adjacency implies that the successful recognition of a preceding construct must occur before the parser can attempt to match a succeeding construct. In the sequence <ab> dog <ef>, the parser will advance to search for string "dog" only if a match to nonterminal <ab> has occurred. Similarly, the string "dog" must be found in the input stream before the parser will advance to search for a match on nonterminal <ef>. An "or" relationship may also exist between groups of constructs and is signified in DML by the "|" symbol. Conceptually, we think of the string "dog" as a single logical entity. In DML, however, each character of a string is considered a separate word. A boolean expression for the outcome of a parse of <ab> cd <ef> would therefore be written: outcome == <ab> & c & d & <ef>.

In a properly written DML syntax specification, each nonterminal appearing on the right side of a ":=" construct must appear exactly once on the left side (as an entry name). Note that terminal symbols never appear in a left part. The implication of this arrangement is that nonterminal symbols do not themselves represent words in the source language but serve as implicit pointers to other productions. Ultimately, this "buck-passing" must terminate in a production(s) specifying a sequence of one or more words (or terminals) in the source language.

Use of the word *word* will remain informal, and I will rely on the reader to infer the appropriate meaning from its context. It could denote a single character (e.g., ASCII "a") or a more complex collection of characters, called a *token* (the reserved work "procedure" in Pascal, for instance). In this way the syntax of complex sequences of words, or *sentences*, may be developed as a sequence of smaller constructs. Each production then represents a *sentential form* in the grammar of the source language. Parsing an input is essentially a process of identifying a correct sequence of sentential forms. Several methods can accomplish this, but I will defer discussion of them to

the section devoted to the parser.

The next point to make is that productions need not be defined in a strictly hierarchical fashion. Although not demonstrated in the simple example of Listing One, DML allows productions to invoke themselves recursively. Consider two productions <a> and <b>:

```
<a>:=<b>;
<b>:=xyz|hij<a>;
```

One of the parsing paths in <b> results in <a> invoking itself by *indirect recursion*. It is important to note, however, that neither parsing path in <b> may begin with a nonterminal symbol <a>. Without this constraint, the indirect recursive invocation of <a> would proceed indefinitely, resulting in stack overflow. The feature of recursive invocation allows for convenient descriptions of highly complex syntactic structures, but also makes it possible to define subtly incorrect syntactic descriptions. In practice, to avoid such pitfalls, you need only apply a few simple rules, which I will enumerate later. More advanced compiler writing systems (e.g., YACC) check for constructs that lead to runaway recursion; the DML compiler does not.

Having given an impression of what nonterminals are and what they do, I'll turn now to terminal constructs and the special forms they may have in DML. Obviously, to parse an input in the source language, the parser must get the input words from somewhere. While you can provide the parser with all the necessary facilities to perform file I/O and maintain its own pointers in the input stream, it is far more common to relegate these more prosaic tasks to the *lexical scanner*, which serves as an interface between the source and the parser. DML takes advantage of this arrangement by pushing part of the responsibility of decoding input onto the scanner, which can do it more efficiently. Consider the two productions below:

```
<character>:=a|A|b|B|c|C ...
  ...x|X|y|Y|z|Z;
<character>:="6";
```

The first specifies that the parser will accept as input an ASCII character and check it against a linked list, or *syntax tree*, of valid ASCII alpha characters for a match. Since the syntax tree (described later) is rather complex, considerable computation might be needed to perform this check. The second definition of <character> employs the special DML syntactic form called the *absolute binary type*. This form (a positive integer enclosed in double quotes) requires the scanner to check the input character against a predetermined list of ASCII characters and, if a match is found, to return to the parser a type value of binary 6. Since the scanner returns both the current symbol and its type (if any), the parser can then pick and choose. In one application, the ASCII character may be needed; in another, the more efficient binary type may be used. This construct results in greater flexibility and efficiency. Absolute binary types may be added or deleted as needed by simple changes to the scanner output routines.

One final terminal construct remains undefined: the *metaconstruct*. DML requires that any occurrence of a metasymbol (a symbol used in defining the syntactic constructs of DML) outside of a legal DML construct must be enclosed in single quotes to identify it as a terminal symbol in the source language. This eliminates ambiguities for the DML compiler.

DML contains four constructs that direct the output from the parser. Each construct represents one data structure that the DML parser is capable of generating. The syntax for the four constructs is illustrated in Table I (page 48). (The actual operation of the DML-compatible parser presented in Listing Two (page 57) must be partially described here as well, since the language and its parser are integrally related.) The most commonly used data type is the *semantic stack*. User-specified output to it is controlled by the *type macro* – an integer enclosed within parentheses preceding the right angle bracket of a nonterminal descriptor (e.g., <abc(123)>). This form causes the specified number to be pushed onto the semantic stack on successful completion of the nonterminal. For example, in the production <wildcard name> in Listing One, the construct <first part (15)> occurs. If the parser successfully identifies a <first part> construct at this point in a parse, it is directed to push the binary value 15 onto the semantic stack. This allows the specification of complex constructs by a single output value.

The second construct available is output to a symbol table; it is represented by a colon following the left angle bracket in a nonterminal descriptor (e.g., <:abc>). All terminal symbols parsed during invocation of the nonterminal will be concatenated in a symbol table buffer. Upon successful completion of the nonterminal, the symbol table will be searched for the parsed word and, if it is not found, the word will be entered. The symbol table entry number is returned to the parser, and a two-number sequence is pushed onto the semantic stack. The first number is a user-defined constant specifying "symbol table entry," and the second is the symbol table entry number. The symbol table utility is not a formal part of the DML runtime system. The DML parser simply calls an external subroutine "dgsntr" and expects a symbol table number to be returned. Any other function desired may be substituted for the symbol table utility by making an appropriate substitution for the routine "dgsntr." (The DML system diskette offered to support this article contains a digital searching symbol table utility.)

The third construct specifies that all correctly parsed terminal symbols be concatenated into a string array. The DML representation is similar to symbol table output with the colon replaced by an exclamation point: <!abc>. A two-number sequence is again pushed onto the semantic stack. The first is a user-defined constant specifying "character string"; the second is an index to the string in the string array.

To avoid potential conflicts between <!xxx> and <:xxx> constructs, the language specification of DML requires that only one be active at any given time; namely, the first one invoked. Consider the example DML productions below:

```
<a>:=<!b><:c>;
<b>:=<:d><!e><:f>;
```

While the ! construct of <b> in production <a> is active, output constructs in <d>, <e>, and <f> are disabled. This is not true of type macros like <xxx(123)>, which are always implemented on successful completion of their corresponding nonterminal.

The final output construct, <abc (xyz)>, does not actually specify a data structure; rather it invokes a subroutine with external entry symbol xyz on successful completion of the enclosing nonterminal. All such subroutines are external to the DML parser, and there lies the secret to the extensibility of DML specifications. These external routines have access to all stacks and pointers that the DML parser maintains via formal parameter passing and global data specifications (common blocks). They may also manipulate user-defined data structures, perform semantic processing, or be modified with no effect on the parser or its syntax specification. Therefore, the scope of parsing actions may be expanded without changing any part of the core DML routines.

DML invokes the specified external routines by calling subroutine "action" given in Listing Five (page 75). This simple routine retrieves the address of the requested routine and vectors to it by appropriately resetting the program counter. The only constraints on the called routines are: (1) they adhere to the parameter passing conventions of the DML parser; and (2) they appropriately set a status flag ("irep") before returning. The DML parser interprets FALSE on return as unsuccessful completion of the current nonterminal.

Frequently a given sequence of syntactic constructs repeats one or more times in succession. DML provides a mechanism for specifying the expected occurrence of such repeats. In Listing

One, observe that a <primary name> is defined by the construct [<name char> (8,1)]. The enclosing brackets imply that the word(s) specified by nonterminal <name char> will repeat a minimum of once and a maximum of eight times. Since <name char> may be absolute binary type 6 (integer), 5 (a-z, A-Z), or 7 (most other printable characters), <primary name> is therefore defined as an alphanumeric string of minimum length one and maximum length eight. Any number of terminal and nonterminal constructs may appear within this *iteration construct*. Further, DML specifies that iteration constructs may be indefinitely nested, although adjacent common left or right brackets are forbidden as shown in Figure 2 (page 50).

The absence of an *iteration macro* construct (m,n) preceding the right bracket implies that no maximum or minimum iteration count is required; i.e., the given construct may appear zero or more times. If either number is assigned a value of zero, the corresponding iteration counter is unspecified.

The DML Syntax Tree

A syntactic description in DML, once made, must be translated into machine-readable form. This section describes the DML syntax tree, a binary tree representation of DML syntax. The idea of using a syntax tree graph to describe syntactic relationships is not new, and most texts on parsing techniques will discuss the basics of this approach. To avoid becoming too verbose, I'll restrict myself to its specific application to the DML language. Henceforth, the words *terminal, nonterminal,* and *node* will be used interchangeably since the binary tree node is the machine equivalent of the terminal and nonterminal syntactic descriptors.

At each step in the parse, there are at most two potential pathways to follow. The DML language requires that, if the current construct is matched, a *successor,* if present, should be tried. If the current construct is not matched, then an *alternate* (signified by "|" in DML), if present, should be tried. The binary tree is ideal for such a representation since each node may have two daughters, which we may label successor and alternate in this application. If no other decisions were needed at each node, a complete syntactic representation could be made using a simple binary tree structure. However, to fully implement DML, as many as ten decisions must be made at each node. This results in a more complex data structure, but it is still essentially a binary tree.

To implement all ten decisions, a total of sixteen bytes would potentially be required at each node. However, very few nodes in a syntax tree graph require a full sixteen bytes, since most do not represent the entire repertoire of DML logical constructs. In practice, the average node needs only four or five bytes, so a considerable amount of space can be saved by including in each node only data elements that are necessary. This approach requires the addition of a status byte, bit-mapped with eight flags. Such a structure makes extracting data elements and hopping about from one node to the other a bit more complicated, but these functions can be accomplished efficiently using routines written in assembler.

Figure 3 (below) compares a DML syntax specification for <unambiguous name> from Listing One with a graphical

Syntax	Action
1. <abc (nn)>	On successful completion of <abc>, push integer nn onto semantic stack
2. <!abc>	Concatenate all terminal symbols invoked by abc into a string; push string location onto semantic stack
3. <:abc>	Concatenate all terminal symbols invoked by abc and enter into a symbol table; push symbol number onto semantic stack
4. <abc (xyz)>	On successful completion of abc, call a subroutine named "xyz"
5. <abc (xyz, nn)>	Do 1 and 4 above
6. <:abc (xyz, nn)>	Do 1, 3, and 4 above
7. <!abc (xyz, nn)>	Do 2, 3, and 4 above

Table I
Output Syntax

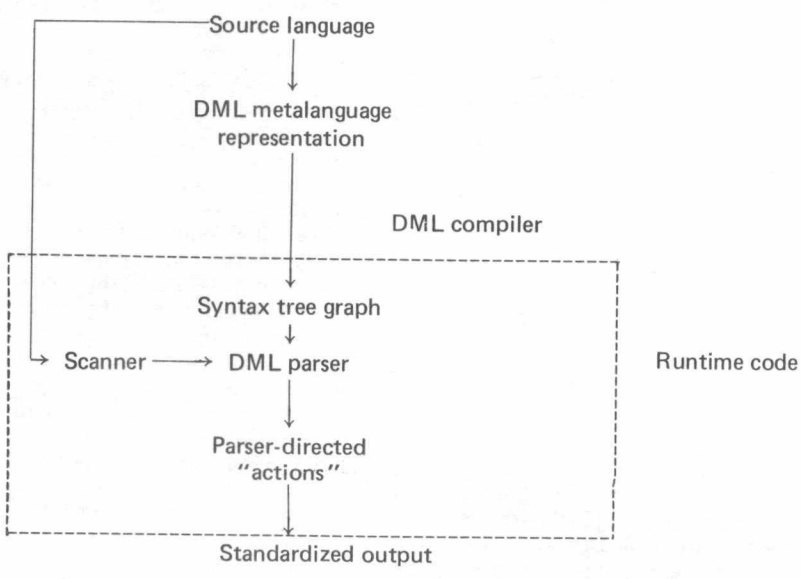


Figure 1.
DML System

representation of a complementary binary tree and hypothetical code generated for it relative to a base address of 1. The complete tree structure generated for DML specifications is rather more complex, but I'll give you a basic impression of how they work for this simplified example. Let me emphasize, however, that it is unnecessary to have any understanding whatsoever of how DML code is generated to use the DML parsing system effectively. Those who aren't interested may wish to skip this section altogether.

Figure 3 illustrates the simplified control structures for a typical production. Presumably some other production invokes <unambiguous name>, causing the parser to jump to the required node. Node I is found to be nonterminal, so the parser again retrieves the address of the requested production <primary name> from the current node (50), suspends processing of the current production, and recursively calls itself to process <primary name>. On return from <primary name>, the parser pursues a successor node if <primary name> was successful and an alternate node if not. Since node I contains no alternate choice, execution of the current production would have to cease on a FALSE return from <primary name>. A successful return causes the parser to pursue the successor node located at 6. The "." found there is compared with the current word from the input stream. If a match occurs, the input pointer is incremented and a successor node, if present, is followed (in this case, <extension> located at 11). Note that nodes II and III form a loop, which could be traversed endlessly were it not for the "maximum" specification located in node II. Each time II is successfully traversed, a counter is incremented and tested against the maximum allowable iteration count. If it is exceeded, the outcome at node II is set FALSE and an alternate is sought. Had a minimum iteration specification been present in node II, it would have been compared to the current iteration counter when the alternate path was traversed, and an error would have been signaled if the requisite number of iterations had not been performed.

A lot of zeros appear in the binary representation in Figure 3. These don't exist in the actual packed syntax tree, so the only way the parser can keep track of whether a given node contains a certain parameter is to consult the corresponding status bit in the flag byte of each node. To keep track of where the parameter resides relative to the flag byte requires that all the flag bits in the current node be considered. The parser doesn't concern itself too much with this arithmetic chore;

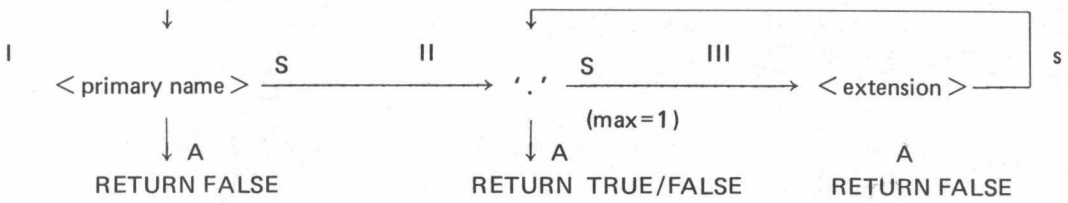
```
(a) <f> := [<a> [<bc> <de> "12"] [<ef> ';' <gh>] ij (3,1)];

(b) <c> := [[<ab> <cd>] ef [<gh> ij <k>] (8,1)];

(c) <c> := [<alt1> ef <alt2> (8,1)];
    <alt1> := [<ab> <cd>];
    <alt2> := [<gh> ij <k>];
```

Figure 2.

(a) is a legal iteration construct while (b) is not. This is not a practical limitation, since constructs such as that shown in (b) can be rewritten to eliminate common adjacent brackets, as shown in (c).

A. < unambiguous name > := < primary name > ['.' < extension > (1,0)];

B.



C.

NODE	ADDRESS	TERMINAL (T) NONTERMINAL (N/T)	INTER-NODE POINTERS SUCCESSOR (S)	INTER-NODE POINTERS ALTERNATE (A)	ITERATION COUNT MAXIMUM	ITERATION COUNT *MINIMUM*
I	1	N/T (50)*	6	0	0	0
II	6	T '.'	11	0	1	0
III	11	N/T (100)*	6	0	0	0

*Nonterminal pointers: 50 = < primary name > ; 100 = < extension >

Figure 3.
Syntax-Tree Representation

it's all taken care of by modules "intarg" and "bytarg" in ddjmutls.mac, shown in Listing Five. Comparing the node flag bytes with the operation of these two simple routines will reveal precisely how the parser is able to navigate the densely packed data types in a syntax tree graph. Note that a lot of flag testing gets done in the process.

As previously stated, this simplified example is not the whole story on syntax tree graph structure and function. Those wishing to pursue a more detailed study will find the DML compiler very helpful. It has a trace mode and logic output feature that, in essence, traces the compiler's "thought processes" in generating syntax trees.

The DML Parser

The DML language specification determines what actions the parser must take at every step in the parse, while the syntax-tree data structure determines how the parsing rules will be transmitted to the parser at runtime. With this groundwork laid, I'll discuss the programs that actually perform the parse.

Formally, the DML parser is a "syntax-tree-directed, recursive descent parser," one of a general class of parsers called "top-down" parsers. The mechanical operation of such parsers is equivalent to the logical development of DML syntaxes: parsing begins with larger, generalized syntactic structures expressed as nonterminals and proceeds "downward" in a recursive fashion to end with terminal symbols in the source language. In general, such parsers are not quite as efficient as another broad class of parsers, termed "bottom-up" parsers. Generating parsing rules for bottom-up parsers is more complicated, and the automated systems that do so are not as easily implemented on microcomputers.

Listing Two (page 57) is a complete listing of the DML parser in RATFOR. This may seem an unlikely choice of languages, so I'll elaborate but make no apologies. RATFOR source is far more readable than Fortran source and therefore is easily translated into more widely used languages such as C or Pascal. Furthermore, to specify the invocation of external actions at compile time requires certain parameter passing features and a flexible ".rel" file format; these are absent from the microcomputer implementations of C with which I'm familiar (BDS, Small-C), and I don't like Pascal. Except for implementing the invocation of actions, conversion to C should be very straightforward.

Table II (page 53) lists the array names and appropriate pointer variables corresponding to the required parser inputs and outputs discussed in general terms earlier. Note that the syntax-tree entry point is also passed to the parser as a formal parameter. This allows the same parser to use different syntax trees at various points during program execution. For example, in a compiler application, the parser might be called at the start of execution to decode the input command string, while later it would be used to parse the input source (cdml, the DML compiler, codes it that way).

It is instructive to note how recursive execution is implemented in a language (Fortran) that doesn't normally support it. The decision to use an auxiliary recursion stack was not one of necessity but of choice, since my runtime memory manager allows recursive subroutine calls in Fortran-80. The reason has to do with economizing stack space during the parsing of deeply nested, recursive constructs. Study the operation of the recursion stack and you will observe that the DML parser saves only those variables actually used at the particular level of recursion (LOR) before descending to the next. In contrast, recursive invocation in Pascal causes all variables local to a procedure to be saved and thus is much more wasteful of memory. Additionally, this method is bound to be faster than a version using "pure recursion" on a Z80/8080 due to the minimal relative addressing capabilities of these microprocessors.

Earlier I stated that DML allows indefinite nesting of iteration constructs "[ ]". Although DML may allow it, the DML parser doesn't support it. The version in Listing Two allows a maximum nesting of four levels. This can easily be expanded by some obvious changes to the code.

The DML parser contains another feature that greatly facilitates the parsing of complex syntaxes: it is capable of indefinite back-up. The importance of this may not be obvious. Suffice it to say that, by careful specification of the parsing rules in DML, you can enable the parser to examine any number of successive words in deciding which path to follow. The practical limit of back-up depends on the ability of the scanner to properly reset the input buffers. For the system implemented in the Listings, intended solely to parse input from the console, no special back-up facilities are necessary. If the input source is a disk file, some facility must be provided to buffer a sufficient number of previously parsed words to cover any request from the parser for back-up.

The general purpose scanner routines in Listing Four (page 68) may be used in almost any application to buffer input to the parser. Routine "scnint" is called to initialize the scanner at the start of execution. Each call to "getsym" causes the next valid token to be returned from the input stream. The scanner always buffers enough previously scanned characters to allow the parser to "back up" at any time. If a back-up is attempted that exceeds the buffering capability of the scanner, an error condition is signaled. Routine "rstscn" is called between calls to the parser. This causes the internal scanner buffer to be reset and maintains the maximum possible back-up capability at all times. Changes in scanner output (absolute binary types) may be made readily by appropriate modifications to the designated block of code.

Indefinite back-up, while a powerful tool in parsing complex structures, does present a problem when you must interpret parsing errors. After error detection, several other parsing paths may be attempted before a global exit occurs. The original syntax error therefore may be covered up by the subsequent abortive parsing paths. No general solution to this problem exists for all grammars. DML deals with it by placing one additional constraint on all grammars that are defined as parsable by the system, a constraint I term "adamant optimism." The frivolous name partly reflects the fact that unfortunately no objective formalisms allow you to predict whether a given grammar will disobey this constraint. The constraint is that *no parsing path may be globally exited successfully unless the input symbol pointer is at a location greater than or equal to the maximum value it has attained during any interim parsing paths.* Simply stated, the parser is not allowed to claim a small victory if it has lost the war.

Study of the actions of variables "backup" and "bakadr" will reveal how this is implemented. Error reporting in the demo program in Listing Three (page 64) "prs.rat," is based on the back-up principle. When a syntax error is detected, the offending character and up to nineteen preceding characters are output as "context:". Following "Current symbol on exit:", the first character contained in the last nonterminal that the parser attempted to match is output. Once familiar with this format, you can identify not only the location of the syntactic error, but also the larger sentential construct that was violated, without resorting to a battery of error messages.

Implementing the Listings

Listing Three is a skeletal parser-driver intended to demonstrate proper implementation of the DML system, using as a source grammar the PIP example in Listing One. It demonstrates the proper method of initializing variables and passing data arrays as formal parameters to the parser. Since all array names are passed as formal parameters, you can allocate them dynamically if you have a suitable method of runtime memory management.

If dynamic memory allocation is not important in your application, they may be statistically dimensioned as in Listing Three. Listings Four and Five contain all necessary support routines.

The RATFOR listings must be precompiled and then compiled using Microsoft's Fortran-80. I use a RATFOR preprocessor from Aspen Software. With very minor modification, the source code should also compile with the CP/M User's Group RATFOR preprocessor. The files are then linked according to the order specified in the leading comments in Listing Three. The scanner presented is for general purpose use, including input from a file. In such applications, the marked code in Listing Three would be replaced by a single call to "scnint" as indicated, which takes care of all scanner initialization.

The M80-compatible assembler file "pipstx.mac" contains the executable syntax tree graph and was prepared especially for this article (normally the ".rel" output from the DML compiler would be linked into the runtime code). The addition of external routine "ques" is meant to serve more of a pedagogical purpose than anything else. The syntax specification in Listing One is written such that, without its "action," we would not know whether the "unambiguous" file name returned was truly unambiguous or contained imbedded ?'s. It examines the "primary name" and possibly "extension" found for the presence of ?'s. If any are found, the corresponding type declarators on the semantic stack are incremented by 100. This demonstrates a simple case where applying an additional heuristic external to the syntax specification can aid in the parsing of syntactic constructs that may otherwise be difficult to express.

Program "prs" prompts with a "*". Any pip command line up to 80 characters long of the form

```
dev:file1.ext =
  dev:file2.ext,dev:file3.ext,...
    ...[OV..]
```

will be parsed. Most wild card and abbreviated forms are supported.

Following a successful parse of a command line, the contents of the semantic stack and string array are displayed. A study of these outputs should be most illuminating. Be aware of one possible confusion: all semantic stack outputs are type "byte" except string array indexes (following the designator 100) and symbol table entry number (following the designator 99). These are type "integer" and are displayed by "prs" as two sequential bytes in 8080-style byte-reversed format (i.e., 512 would be displayed as 00 02).

The syntax as given contains two "errors"; that is, the specified syntax does not parse command lines entirely as we might anticipate. I left them in because they demonstrate how subtle changes in syntax can affect the parser. These two conditions are, first, that ambiguous file specifications of the form "filename.*" are interpreted as syntactically incorrect, while "*.ext" and "*.*" are parsed correctly. The problem is corrected by reversing the order of <unambiguous name> and <wild card name> in <file specifier>. Second, all option specifiers [ABC...] must be in upper case, while all other entries may be in either upper or lower case. This is due to the restrictive definitions in <nletter> and <letter>.

DOs and DON'Ts

Syntax-tree graph representations of DML syntaxes are sufficiently easy to generate and modify that trial-and-error is not an intolerable method of learning. Still, certain problem situations should be anticipated and thus avoided.

Most difficulties arise from inappropriate use of iteration constructs of the type [<x>'''<z>]. Be aware that upon detection of an unmatched symbol in the input stream, the parser resets the runtime pointers to the values saved during processing of the *most recent* nonterminal. Consider the two alternative formulations for <value>:

```
<value>:=
  [<arg><u><ments>]|<parameter>;
```

and

```
<value>:=
  [<construct>] | <parameter>;
<construct>:=
  <arg><u><ments>;
```

Assume the intent in both cases is the same: to determine whether <value> is a series of "arguments" or a "parameter." Only the second formulation correctly expresses this intent in all cases. The problem with the first is revealed by considering the outcome of failing to match the nonterminal <u>. DML requires that an *alternate* path be followed on failure. Since no alternate path out of <u> exists (no "|" symbol following <u>), a global failure of production <value> occurs *before* <parameter> is tested. In the second formulation, global failure of <value> does not occur before <parameter> is tried, regardless of where an error occurs in <construct>. There is

Variable Name	Description
sb	Symbol buffer – stores the current symbol being parsed in response to an active ":" flag.
11	Index into sb. Always points to the first free location. This variable is local to parse.
sbptr	Saves the value of 11 for possible back-up.
ss	Symbol string array to hold string output of the parser.
jj	Index into ss. Always points to first free location.
ssptr	Saves the value of jj for back-up.
type	Semantic stack array.
ii	Index into type. Always points to first free location.
typtr	Saves the value of ii for back-up.
kk	Index into the input source. Always points to the next unmatched character.
ismptr	Saves the value of kk for back-up.
st	Symbol table array.
node	Syntax tree graph array.
cndadr	Index into array "node" of the currently active node.
iter	Status of currently active iter'n macros. Structure: iter: (1) (2) (3) (4) (5) # active macros, cndadr, max# iter'ns, min# iter'ns, current# iter'ns Structure (2)-(5) repeats a maximum of three times in the current version.

Table II
Principal Variables in parse.rat

nothing wrong with the first formulation per se; just be aware that the grammatical rule it enforces is more stringent. In conclusion, if subsequent alternates past an *iteration* construct should be tested, the parser must be able to unambiguously select the appropriate parsing path by looking at the first construct in the iteration construct. When in doubt, reformulate as in the second example above.

Another potential problem exists with both of the above formulations. Recall that an iteration construct containing no iteration macro (m,n) specifies zero or more iterations. Therefore, the first alternate will always match in both formulations, and again <parameter> will never be tested. The solution is to put indefinite iteration constructs last in a list of alternates or to reformulate them as [<construct>(0,1)].

Avoid ambiguity between absolute binary types and terminal symbols in the source language. For example, if an absolute binary type value of 65 were used, the parser could not distinguish it from ASCII "A." The set of terminal symbols in the source grammar and the set of defined absolute binary types must be disjoint.

Finally, do not use any left recursive constructs of the form:

```
<a>:=<a>bcd|<def>|<gh>;
```

or

```
<a>:=<def>|<gh>|<a>bcd;
```

The result is indefinite recursive invocation of <a> until stack overflow occurs. It is possible to show that such forms can be rewritten as

```
<a>:=<s>[bcd];
<s>:=<def>|<gh>;
```

with the added benefit of greater efficiency due to elimination of the recursive form.

Availability of the DML System

The code provided in this article is sufficient to produce a working version of the DML parser so the reader can explore its capabilities. To exploit its full potential, the diskette I am making available to support this article is indispensible. The distribution diskette is configured for Z80, CP/M 2.x. It contains all the listings given here as well as the RATFOR precompilations, eliminating the need for a RATFOR precompiler. It also contains "dgsntr," a symbol table utility based upon the digital searching method; a complete representation of the DML metalanguage written in DML; and "cmdl," the DML compiler that produces syntax tree graphs from DML specifications. It outputs code in Microsoft-compatible ".rel" format, machine-independent straight binary, and a special trace-mode logic format. It compiles DML specifications at the rate of approximately 60 productions per minute. Several other support routines are also included to aid in developing and debugging DML specifications.

You may obtain these files by sending a stamped, self-addressed diskette mailer, an 8-inch diskette, and $5.00 (please, no cardboard, Scotch tape, etc.). Those who send no provision for return mailing and/or no diskette, send $25.00. I can also support a number of 5¼-inch formats; write and inquire.

DDJ

(Listings begin on page 56)

DML Parser (Text begins on page 46)

Listing One

```
%       pipstx.stx - 3/15/83
DML representation of a partial command syntax for CP/M
version 2.2, peripheral interchange untility, pip.com.
For brevity, input is assumed to be in upper case.
Accepts command lines of the form:
A:FILE1.EXT=B:FILE2.EXT,C:FILE3.EXT..[ATnOV..]
Supports most wild-card formats.
%

$ pipstx ;

<command> := [ <dev(20)> (1,0) ] <file specifier(10)> '='
     [ <dev(20)> (1,0) ] <file specifier(10)>
     [ ',' [ <dev(20)> (1,0) ] <file specifier(10)> ]
       [ '[' [ <!option> (0,1) ] ']' (1,0) ] ;

<file specifier> := <unambiguous name (ques,11)> |
    <wildcard name(12)> ;

<unambiguous name> := <!primary name(13)>
                [ . <!extension(14)> (1,0) ] ;

<wildcard name> := <first part(15)> | <second part(16)>
    | <both(17)> ;

<first part> := * . <!extension> ;

<second part> := <!primary name> . * ;

<both> := * . * ;

<primary name> := [ <name char> (8,1) ] ;

<extension> := [ <name char> (3,1) ] ;

<option> :=  <nletter> [ "6" (1,0)] | <letter> ;

<dev> := <!devchar> ':' ;

<devchar> := "5" ;

<nletter> := D | G | N | P | T ;

<letter> := B | E | F | H | I | L | N | O |
     Q | R | S | U | V | W | Z ;

<name char> := "5" | "6" | "7" ;

% Absolute binary types defined:
"5" ==>  a-z or A-Z
"6" ==>  0 - 9
"7" ==>  all other printables except < > . , : = * [ ]
"8" ==>  < > . , ; : = * [ ]
%
```

End Listing One

DML Parser (Listing continued, text begins on page 46)

Listing Two

```
# ddjpars2.rat - (C) 1983, David J. Harry
# for non-commercial use only

     define(FALSE,.false.)
     define(TRUE,.true.)
     define(NUL,0)
     define(HEAPLENGTH,17)
     define(ADDRMASK,z'7fff')
     define(EXTMASK,z'8000')

# type declarators for the semantic stack
     define(SYMTYPE,99)#user defined
     define(STRINGTYPE,100) #user defined

# syntax tree node flag values
     define(ALTERNATE,128)
     define(SUCCESSOR,64)
     define(EXCLFLAG,32)
     define(TYPFLAG,16)
     define(MAXFLAG,8)
     define(MINFLAG,4)
     define(TERMINAL,2)
     define(COLONFLAG,1)

#global flag values
     define(GLOBALSTRING,32)
     define(GLOBALSYMBOL,1)

# iteration construct constants
     define(HDRNODE,Z'C0')
     define(ITERCOUNT,3)
     define(MAXCOUNT,1)
     define(MINCOUNT,2)
     define(NUMACTIVE,1)

#:#
#:#==========================>>> PARSE  <<<==========================
#:#
   subroutine parse(node,ss,st,type,match)

     implicit integer (a-z)

     byte node,ss,st,sb,type,data,maxv,minv,ntyp,
       maxerr,minerr,symerr,ptr,eoferr,succ,alt,extrn

     logical max,min,match,pexclm,pcolon,irep,peof,iret,goutp

     dimension node(2),ss(2),st(2),type(2),iter(21),
       sb(25),ptr(17)
     common /pointr/ ii,jj,kk,cndadr,cursym,curtyp

# force sequential storage ---------->
     equivalence (ptr(1),pexclm),(ptr(2),pcolon),(ptr(3),goutp),
```

```
        (ptr(4),max),(ptr(5),min),(ptr(6),ssptr),
        (ptr(8),ismptr),(ptr(10),typtr),(ptr(12),sbptr),
        (ptr(14),savsym),(ptr(16),savtyp)
# ------------> as required by push/pop.

      data data,succ,alt,maxv,minv,ntyp,extrn/0,1,2,3,4,5,6/,
        maxerr,minerr,eoferr,symerr/20,21,22,23/
# disable symbol table initialization
      init=FALSE
# initialize the local backup pointer
      backup=0
#initialize the global output flag
      goutp=FALSE
#initialize the premature EOF flag
      peof=FALSE
# initialize the local symbol buffer pointer
      ll=2
# initialize the recursion stack
      iret=FALSE; call rpush(iret,1); iret=TRUE

# zero the iteration construct count
40    iter(NUMACTIVE)=0
      do i=2,18,4
iter(i)=0# zero all node pointers
# initialize the max & min flags for the current node
50    max=(node(cndadr)&MAXFLAG)!=FALSE
      min=(node(cndadr)&MINFLAG)!=FALSE
      if ((node(cndadr)&HDRNODE)==HDRNODE)
        { # update the iter'n construct counters
if (iter(NUMACTIVE)==0)
      j=2
else
  {
      j=(iter(NUMACTIVE)-1)*4+2
      if (iter(j)==cndadr)
goto 100 # current node iter'n construct already active
      j=j+4
  }
iter(j)=cndadr# activate a new iter'n construct
if (max)
      iter(j+MAXCOUNT)=bytarg(node(cndadr),maxv)
else
      iter(j+MAXCOUNT)=0
if (min)
      iter(j+MINCOUNT)=bytarg(node(cndadr),minv)
else
      iter(j+MINCOUNT)=0
iter(j+ITERCOUNT)=0
iter(NUMACTIVE)=iter(NUMACTIVE)+1
        } # end if(----==HDRNODE)

# determine if current node is terminal or non-terminal
100   if ((node(cndadr)&TERMINAL)!=FALSE)
        { # current node is TERMINAL
```

(Continued on next page)

DML Parser (Listing continued, text begins on page 46)

Listing Two

```
pexclm=FALSE; pcolon=FALSE
curdat=bytarg(node(cndadr),data)
if (curdat==NUL)
  {# fall thru a 'null' node
    if (match)
goto 300# follow successor path
    else
goto 500# follow alternate path
  }
if (!(curdat==curtyp | curdat==cursym))
  {
    match=FALSE; goto 400# further test unsuccessful outcome
  }
match=TRUE
case
  {# process any active global flags
    (goutp==GLOBALSTRING)
      { # current char. to string array
ss(jj)=cursym; jj=jj+1
      }
    (goutp==GLOBALSYMBOL)
      { # char. to symbol buffer
sb(ll)=cursym; ll=ll+1
      }
  }
call getsym(irep)# get next char. to parse
if (! irep)
  {
    if (peof)
      {
match=FALSE; call error(eoferr); return
      }
    cursym=NUL# give the scanner one more chance
    curtyp=NUL; peof=TRUE
  }
      } # end if TERMINAL
    else
      { # current node is non-TERMINAL
# save ptrs. for possible back-up
typtr=ii; ismptr=kk; ssptr=jj; sbptr=ll
savsym=cursym; savtyp=curtyp
# set local output flags preserving global precedence
pexclm=((node(cndadr)&EXCLFLAG | goutp&GLOBALSTRING) !=@
  FALSE)&((goutp&GLOBALSYMBOL) != GLOBALSYMBOL)
pcolon=((node(cndadr)&COLONFLAG | goutp&GLOBALSYMBOL) !=@
  FALSE)&((goutp&GLOBALSTRING) != GLOBALSTRING)
if ((node(cndadr) & TYPFLAG)!=FALSE)
  {# push active type declarator on semantic stack
    type(ii)=bytarg(node(cndadr),ntyp); ii=ii+1
  }
if ( goutp==FALSE)# global flag disables local flags
  {# process any active local output flags
    case
```

```
      {
(pcolon)
  {# push symbol type and slot for symbol number
    type(ii)=SYMTYPE; ii=ii+1
    savnum=ii; ii=ii+2# save location for symbol #
  }
(pexclm)
  {# push 'string' type and slot for string address
    type(ii)=STRINGTYPE;ii=ii+1
    savnum=jj# index of current string base
    call intput(type(ii),jj); ii=ii+2; jj=jj+1
  }
      } # end case
  }# end if(!goutp)
# prepare to "recursively" descend
call rpush(pexclm,HEAPLENGTH); call rpush(cndadr,2)
# stack the iter'n array only if active
if (iter(NUMACTIVE)!=FALSE)
  {
    j=iter(NUMACTIVE)*8; call rpush(iter(2),j)
  }
# must save the 1st element for correct 'pop'
call rpush(iter(NUMACTIVE),2)
# set the global flag fo. the next LOR
goutp=pexclm&EXCLFLAG | pcolon&COLONFLAG
# get the addr. of the header node of next production
cndadr=intarg(node(cndadr),data)
# mask off any 'external' flags
cndadr=cndadr & ADDRMASK
# "call parse"
call rpush(iret,1)
goto 40

# "return" here from lower LOR'S
200call rpop(iter(NUMACTIVE),2)
if (iter(NUMACTIVE)!=FALSE)
  {
    j=iter(NUMACTIVE)*8; call rpop(iter(2),j)
  }
call rpop(cndadr,2); call rpop(pexclm,HEAPLENGTH)
      } #end else NON-TERMINAL

# test for success of current and lower LOR'S
    if (match)
      { # match is TRUE
if (max | min)
  {# process current iter'n construct
    j=(iter(NUMACTIVE)-1)*4+2
    iter(j+ITERCOUNT)=iter(j+ITERCOUNT)+1
 # check for overflow of iter'n max count
    if (max & iter(j+ITERCOUNT)>iter(j+MAXCOUNT))
# set only one option:
```

(Continued on next page)

DML Parser (Listing continued, text begins on page 46)

Listing Two

```
      {
call error(maxerr)
# signal a failure (fatal error)
match=FALSE
goto 450
      }
  }# end if(max ! min)
# invoke 'actions' for non-terminals      .
if (((node(cndadr) & TERMINAL) == FALSE) &
  (intarg(node(cndadr),data) & EXTMASK) != FALSE)
  {
    j=intarg(node(cndadr),extrn)
# alter parameter passing as appropriate
    call action(j,irep,st,ss,type)
    if (!irep)
      {
match=FALSE
# implement only one 'goto'
goto 400# reset & continue parse
#goto 700 # global return on error
      } # end if(!irep)
  }# end if( test for action..
if (goutp==FALSE)
  {# test for end of current output token
    case
      {
(pexclm)

  {# terminate the active string
    i=jj-savnum-1# length of current string
    ss(savnum)=i# insert length at string base
  }
(pcolon)
  {# terminate the active symbol
    sb(1)=ll-2# length in symbol buffer
    irep = TRUE # enable table entry

# *** call to external symbol table utility
#     dummied out for demo use
#call dgsntr(st,sb,i,init,irep) # find/enter current symbol
    i=-1
# ***

    if (! irep)
      { # error during symbol table routine
call error(symerr); i=0
      }
    call intput(type(savnum),i) # enter symbol # in the semantic stack
    ll=2# reset the symbol buffer ptr.
  }
      } # end case
  }# end if(!goutp)
300if ((node(cndadr)&SUCCESSOR)==SUCCESSOR)
```

```
  {# follow the successor node
    cndadr=intarg(node(cndadr),succ); goto 50
  }
else# "return"
  {
    goto 600
  }
      } # end if(match)
    else
      {# match=FALSE - further test outcome at current node
400if ((node(cndadr)&HDRNODE)==HDRNODE)
  {# node is an iter'n header node
    j=(iter(NUMACTIVE)-1)*4+2
    if (!min | min&iter(j+MINCOUNT)<=iter(j+ITERCOUNT))
match=TRUE# min achieved or none required
450     iter(NUMACTIVE)=iter(NUMACTIVE)-1# decr. construct count
    iter(j)=0# zero header node ptr.
  }# end if( - ==HDRNODE)
if (kk>backup)
  {# update the backup ptr.
    backup=kk; bakadr=cndadr
  }
if ((node(cndadr)&TERMINAL)==FALSE)
  {# reset ptrs. for unsuccessful non-terminal node
    ii=typtr; jj=ssptr; kk=ismptr
    ll=sbptr; cursym=savsym; curtyp=savtyp
  }
500if ((node(cndadr)&ALTERNATE)==ALTERNATE)
  {# follow the alternate node ptr.
    cndadr=intarg(node(cndadr),alt); goto 50
  }
      } # end else (match)
# "return" routine
600 call rpop(iret,1)
    if (iret)
      goto 200
# check for hidden parser failure before global return
700 if (backup<=kk & match)
      return
# report location of parser error
    match=FALSE; kk=backup; cndadr=bakadr
    return

    end
```

End Listing Two

Listing Three

```
#- prs.rat - 3/21/83
#  skeletal parser driver.
#  loader order for 180: prs,ddjpars2,pipstx,ddjrutls,ddjmutls

    define(TRUE,.true.)
```

(Continued on page 66)

DML Parser (Listing continued, text begins on page 46)

Listing Three

```
      define(FALSE,.false.)
      define(STACKSIZE,1000)

#:#
#:#=============================>>> PRS     <<<=============================
#:#
  program prs
      implicit integer (a-z)
      byte type,ss,st,buffer,dml,fcb,stk,fcb
      logical irep,match,stop,eof
      external pipstx

      dimension type(500),ss(500),st(2),stk(STACKSIZE)

      common /pointr/ ii,jj,kk,cndadr,cursym,curtyp@
        /scanr/ fcb(36),buffer(256),eof,jbase,top,stop

10    write(1,2100)
2100  format(' *')
      read(1,1000)(buffer(i),i=1,80)
1000  format(80a1)
# get 1st valid chr.

# When input is via a disk file, the blocked code
# is replaced by a single call to 'scnint.rat'
# call scnint(flname,unit,irep)
# test 'irep' for TRUE return

# ****************
      stop=FALSE; eof=TRUE
      top=81; kk=1; jbase=0
# ****************

      call getsym(match)
      if (! match)
          goto 10
      call rststk(stk,STACKSIZE)  # reset recursion stack
      cndadr=1; ii=1; jj=1        # initialize pointers
      call parse(pipstx,ss,st,type,irep)
      if (irep)
        {                         # process successful parse of input
          j=ii-1
          write(1,2110)(type(i),i=1,j)
2110      format(' ',10i6)
          if (jj != 1)
            {
              i=1
              while (i < jj)
                {
                  m=ss(i); itop=i+m; i=i+1
                  write(1,2080)(ss(k),k=i,itop)
2080              format(' ',50a1)
                  i=itop+1
```

```
        }                       # end while (i < jj)
          }                      # end if (j != 1)
# insert this call for file I/O and test
# 'irep' for TRUE return
# call scnrst(irep)

      }                         # end if(irep)
    else
      {                         # report erroneous production
        write(1,2090)
2090    format(' !! Syntax error !!')
        j=kk-1                  # position of last chr. parsed
        if (j <= 0)
            j=1
        m=j-20
        if (m < 1)
            m=1
        write(1,2150)cursym,(buffer(i),i=m,j)
2150    format(' Current symbol on exit: ',1a2,'  context: ',25a1)
      }                         # end else (irep)
# write blank lines between outputs
    write(1,2001)
2001 format(/)
    goto 10

    end
```

End Listing Three

Listing Four

```
# - ddjrutls.rat - 3/17/83

    define(PUSHERROR,50)
    define(POPERROR,51)

#:#
#:#=============================>>> RPUSH  <<<=============================
#:#
  subroutine rpush(array,n)
    byte irep
    call push(array,n,irep)
    if (!irep)
      {
call error(PUSHERROR)
stop
      }
    return
    end

#:#
#:#=============================>>> RPOP   <<<=============================
#:#
  subroutine rpop(array,n)
    byte irep
    call pop(array,n,irep)
    if (!irep)
      {
```

(Continued on next page)

DML Parser (Listing continued, text begins on page 46)

```
call error(POPERROR)
stop
      }
    return
    end

#- scnint.rat / scnrst.rat / getsym.rat

# Scan an input source file and return tokens and token types to the
# calling routine.  The buffer always maintains the previous 128
# characters (if possible) to allow for back-up.
#scnint.rat is called to initialize the scanner and read in the
# first (two/one) logical sector(s).
#scnrst.rat should be called between calls to the parser.  In this
# way, the input buffer pointer will be properly reset and the maximum
# possible back-up capability maintained at all times.No other routines
# should reset the buffer pointer (kk) either between or during calls to
# the parser, or unpredictable operation may result.  If the parser
# forces an input symbol back-up greater than the maximum permissable
# value (currently 128 total characters, INCLUDING ignore characters),
# an error is flagged and the scanner returns match=FALSE.

    define(FALSE,.false.)
    define(TRUE,.true.)
    define(INVALID,-1)
    define(NOTHER,9)
# defined symbol types
    define(ALPHA,5)# user defined
    define(INTEGER,6)#  "
    define(MISC,7)#  "
    define(OTHER,8)#"
#- scnint.rat - initialize the scanner buffer ----------------------------

#:#
#:#=============================>>> SCNINT <<<=============================
#:#
  subroutine scnint(flname,unit,irep)
    implicit integer (a-z)
    byte buffer,fcb
    logical eof,irep,stop
    common/pointr/ii,jj,kk,cndadr,cursym,curtyp@
      /scanr/fcb(36),buffer(256),eof,jbase,top,stop
    data eoferr/22/

    call makfcb(fcb,flname,unit)
    call opnfil(fcb,irep)
    if (!irep)
goto 800
    call setdma(buffer(1))
    call rdseq(fcb,irep)
    if (!irep)
goto 800
    call setdma(buffer(129))
    call rdseq(fcb,irep)
    if (!irep)
```

```
      {
top=129; eof=TRUE
      }
    else
      {
top=257; eof=FALSE
      }
    stop=FALSE
    kk=1
    jbase=0
    irep=TRUE
    return
800 call error(eoferr)
    irep=FALSE
    return
    end

#- scnrst - reset input source to base of buffer ----------------------------

#:#
#:#==========================>>> SCNRST <<<==========================
#:#
  subroutine scnrst(match)
    implicit integer (a-z)
    byte buffer,fcb
    logical eof,irep,stop,match
    common/pointr/ii,jj,kk,cndadr,cursym,curtyp@
      /scanr/fcb(36),buffer(256),eof,jbase,top,stop

    if (stop)
goto 800
    cur=kk-jbase
    nmove=top-cur
    call blkmov(buffer(cur),buffer(1),nmove)
    top=nmove+1
    if (nmove<=128)

      {
call setdma(buffer(top))
call rdseq(fcb,irep)
if (!irep)
    eof=TRUE
else
    top=top+128
      }
    jbase=0
    kk=1
    match=TRUE
    return
800 match=FALSE
    cursym=INVALID
    curtyp=INVALID
    return
    end

#- getsym.rat - scan the input stream ---------------------------------------
```

(Continued on next page)

DML Parser (Listing continued, text begins on page 46)

Listing Four

```
#:#
#:#=============================>>> GETSYM <<<=============================
#:#
  subroutine getsym(match)
    implicit integer (a-z)
    byte buffer,fcb,curchr,metsym
    logical eof,match,irep,stop
    dimension metsym(NOTHER)
    common/pointr/ii,jj,kk,cndadr,cursym,curtyp@
      /scanr/fcb(36),buffer(256),eof,jbase,top,stop
    data metsym/60,62,46,44,58,61,42,91,93/

    match=FALSE
# check for illegal back-up or eof
    if (kk<jbase | stop)
goto 800
    while (!match)
      {#. keep trying until eof or legal token is found
cur=kk-jbase
curchr=buffer(cur)
kk=kk+1
cur=cur+1
if (cur >= top)
  {# test for end-of-buffer
    if (!eof)
      { # move buffer down
base=cur-128
call blkmov(buffer(base),buffer(1),128)
jbase=kk-129# readjust offset of buffer base
call setdma(buffer(129))
call rdseq(fcb,irep)
if (irep)
    top=257
else
  {
    top=129
    eof=TRUE
  }
      } # end if(!eof)
    else#no more tokens to scan
      {
stop=TRUE; goto 800
      }
  }# end if(cur >= top)
case

# *** this section may be altered as needed for
#     the particular application
  {# test for valid token
    (curchr<33 | curchr>126)
;# invalid char. - ignore it
```

```
    (curchr>47 & curchr<58)
      { # INTEGER token found
match=TRUE; curtyp=INTEGER; cursym=curchr
      }
    (curchr>64 & curchr<91 | curchr>96 & curchr<122)
      { # ALPHA token found
match=TRUE; curtyp=ALPHA; cursym=curchr
      }
    (otherwise)
      {
match=TRUE; cursym=curchr
do i=1,NOTHER
  {# determine type
    if (curchr == metsym(i))
      {
curtyp=OTHER
return
      }
  }# end do
curtyp=MISC
      } # end otherwise
  }# end case
# *****

      } # end while
    return
800 cursym=INVALID
    curtyp=INVALID
    return
    end
# skeletal error reporting routine
#:#
#:#=============================>>> ERROR  <<<=============================
#:#
  subroutine error(type)
    byte type
    write(1,2000)type
2000 format(' Error! Type number: ',i4)
    return
    end

# ques.rat - simple routine to demonstrate the use
# of externally referenced 'actions'.
# Test for imbedded '?' in (un)ambiguous filenames. Invoked
# in non-terminal <unambiguous name> in production
# <file specifier>. Note 'ques' must ignore 1st formal parameter.

    define(QUESTIONMARK,63)
    define(AMBIGUOUS,100)
    define(PRIMARYNAME,13)
    define(TRUE,.true.)
```

(Continued on next page)

DML Parser (Listing continued, text begins on page 46)

Listing Four

```
#:#
#:#==================================>>> QUES    <<<=====================================
#:#
  subroutine ques(dummy,irep,st,ss,type)
    implicit integer (a-z)
    byte st,ss,type
    logical irep
    dimension st(2),ss(2),type(2)
    common /pointr/ii,jj,kk,cndadr,cursym,curtyp

    k=ii
    do tot=1,2
      {
i=intget(type(k-2)); j=i+ss(i); n=i+1
do i=n,j
  {
    if (ss(i) == QUESTIONMARK)
      {
type(k-4)=type(k-4)+AMBIGUOUS
goto 100
      }
  }
100k=k-4
if (type(k-4) != PRIMARYNAME)
    goto 200
      } # end do
200 irep=TRUE
    return
    end

# dummy routines for demo version - distribution diskette
# contains all these routines. Program 'prs' is configured
# such that none of these routines will be called.

    subroutine makfcb# CP/M function # 22
    return
    end
    subroutine opnfil# CP/M function # 15
    return
    end
    subroutine setdma# CP/M function # 26
    return
    end
    subroutine rdseq# CP/M function # 20
    return
    end
  subroutine blkmov # byte oriented block-move utility
    return
    end
```

End Listing Four

Listing Five

```
;- ddjmutls.mac - 3/17/83 - written for Microsoft's M80

;       equates
        DATA    EQU     0
        SUCC    EQU     1
        ALT     EQU     2
        MAX     EQU     3
        MIN     EQU     4
        TYPE    EQU     5
        TORNT   EQU     2
        SUCCF   EQU     40H
        ALTF    EQU     80H
        MAXF    EQU     8
        MINF    EQU     4
        TYPF    EQU     16
        TRUE    EQU     OFFH
        FALSE   EQU     0

        GLOBAL  PUSH,POP,RSTSTK,INTARG,BYTARG,ACTION,INTPUT,INTGET

        .Z80
        CSEG

BYTARG:
        XOR     A
        LD      (RTN),A
        JR      AR0
INTARG:
        LD      A,1
        LD      (RTN),A
AR0:
        LD      A,(DE)          ; GET PARM
        LD      B,A
        LD      C,(HL)          ; GET NODE FLAG
        INC     HL              ; PNT. TO DATA/PTR
        XOR     A
        CP      B
        JR      Z,DONE          ;JUMP IF DATA/PTR REQUESTED
        LD      A,TORNT
        AND     C
        JR      NZ,AR1
        INC     HL
AR1:
        INC     HL              ; HL PNTS TO SUCC
        DEC     B
        JR      Z,DONE          ;JUMP IF SUCC REQUESTED
        LD      A,SUCCF
        AND     C
        JR      Z,AR2
        INC     HL
        INC     HL              ; HL PNTS TO ALT
AR2:
        DEC B
```

(Continued on next page)

DML Parser (Listing continued, text begins on page 46)

Listing Five

```
        JR      Z,DONE          ;JUMP IF ALT REQUESTED
        LD      A,ALTF
        AND     C
        JR      Z,AR3
        INC     HL
        INC     HL              ; HL PNTS TO MAX
AR3:
        DEC     B
        JR      Z,DONE          ;JUMP IF MAX REQUESTED
        LD      A,MAXF
        AND     C
        JR      Z,AR4
        INC     HL              ; HL PNTS. TO MIN
AR4:
        DEC     B
        JR      Z,DONE          ; JUMP IF MIN REQUESTED
        LD      A,MINF          ; MUST WANT TYPE
        AND     C
        JR      Z,AR5
        INC     HL              ; HL PNTS TO TYPE
AR5:
        DEC     B
        JR      Z,DONE          ; JUMP IF TYPE REQUESTED
        LD      A,TYPF
        AND     C
        JR      Z,DONE
        INC     HL
DONE:           ; GET THE ARGUMENT
        LD      E,(HL)
        LD      A,(RTN)
        OR      A
        JR      Z,AR6
        INC     HL              ; RETURN AN INTEGER
        LD      D,(HL)
        EX      DE,HL
        RET
AR6:            ; RETURN A BYTE BOTH AS AN INTEGER AND A BYTE
        LD      D,0
        EX      DE,HL
        LD      A,L
        RET

RTN:    DEFB    0

ACTION:         ; vector to an externally referenced routine.
        LD      A,(HL)
        INC     HL
        LD      H,(HL)
        LD      L,A
        JP      (HL)    ; and we're gone...
```

```
;------ RSTSTK.MAC / PUSH.MAC / POP.MAC ------------------------------------

;       Stack functions are initialized by calling RSTSTK:

;       CALL RSTSTK(BASE,NBYTES)

;       BASE points to a byte array of length NBYTES, to be used for the
;       stack.
;       Push variables onto the stack.  Check for stack overflow before
;       the transfer.  FORTRAN calling format:

;       CALL PUSH(ARRAY,NBYTES,IREP)

;       Pop variables from the stack. Check for stack underflow before the
;       the transfer.  FORTRAN calling format:
;       CALL POP(ARRAY,NBYTES,IREP)

;       NBYTES is the number of bytes to transfer to/from ARRAY and is
;       assumed to be type 'INTEGER'. IREP is a logical parameter that is
;       returned FALSE on stack under/over flow;  TRUE on normal return.
;--------------------------------------------------------------------------

RSTSTK:
        LD      (BASE),HL       ; SAVE BASE ADDR. OF STACK
        LD      (FSTFRE),HL     ; INIT. STACK POINTER
        EX      DE,HL
        LD      A,(HL)
        INC     HL
        LD      H,(HL)
        LD      L,A             ; SAVE STACK SIZE
        LD      (STKSIZ),HL
        XOR     A
        ADC     HL,DE           ; CALC' TOP-OF-STACK
        DEC     HL
        LD      (TOP),HL
        RET

PUSH:
        PUSH    BC              ; SAVE 'IREP' ADDR.
        PUSH    HL              ; SAVE ARRAY ADDR.
        EX      DE,HL
        LD      E,(HL)          ; GET # TO XFER
        INC     HL
        LD      D,(HL)
        PUSH    DE              ; SAVE #
        LD      HL,(FSTFRE)
        XOR     A               ; CLEAR CY FLAG
        ADC     HL,DE
        DEC     HL              ; (FSTFRE+#-1)
        LD      DE,(TOP)
        SBC     HL,DE           ; (FSTFRE+#-1)-TOP
        JR      C,P1
        POP     HL              ; RESTORE PROGRAM STACK BEFORE EXIT
        POP     HL
        POP     HL              ; GET 'IREP' ADDR.
        LD      (HL),FALSE
        RET
```

(Continued on next page)

DML Parser (Listing continued, text begins on page 46)

Listing Five

```
P1:
        POP     BC                      ; BC = #
        POP     HL                      ; HL = ARRAY ADDR.
        LD      DE,(FSTFRE)
        LDIR
        LD      (FSTFRE),DE             ; UPDATE FIRST FREE LOCATION
        POP     HL
        LD      (HL),TRUE
        RET

POP:
        PUSH    BC
        PUSH    HL                      ; SAVE ARRAY ADDR.
        EX      DE,HL
        LD      E,(HL)                  ; GET # TO XFER
        INC     HL
        LD      D,(HL)
        PUSH    DE                      ; SAVE # TO XFER
        .
        LD      HL,(FSTFRE)
        XOR     A
        SBC     HL,DE                   ; FSTFRE - #
        PUSH    HL                      ; SAVE POSSIBLE NEW FIRST FREE LOCATION
        LD      DE,(BASE)
        SBC     HL,DE
        JR      NC,P01
        POP     HL                      ; RESTORE PROGRAM STACK BEFORE EXIT
        POP     HL
        POP     HL
        POP     HL
        LD      (HL),FALSE
        RET
P01:
        POP     HL                      ; HL = STACK LOCATION
        POP     BC                      ; BC = #
        POP     DE                      ; DE = ARRAY ADDR.
        LD      (FSTFRE),HL             ; UPDATE FIRST FREE LOCATION
        LDIR
        POP     HL
        LD      (HL),TRUE
        RET


STKSIZ:         DEFW    0
FSTFRE:         DEFW    0
BASE:           DEFW    0
TOP:            DEFW    0

INTPUT: ; subroutine to 'put' an integer argument
        LD      A,(DE)
        LD      (HL),A
        INC     HL
        INC     DE
```

```
        LD      A,(DE)
        LD      (HL),A
        RET

INTGET: ; integer function to 'get' an integer argument
        LD      A,(HL)
        INC     HL
        LD      H,(HL)
        LD      L,A
        RET

        END
```

End Listing Five

Listing Six

```
; pipstx.mac - assembler version of pipstx.stx syntax tree

.z80
cseg
global pipstx
external ques

pipstx:
defb 0D8h,0DBh,00h,01h,00h,0Ah,00h,01h,14h,42h,00h,0Eh,00h,50h
defb 61h,00h,14h,00h,0Ah,42h,3Dh,18h,00h,0D8h,0DBh,00h,18h,00h,21h,00h
defb 01h,14h,42h,00h,25h,00h,50h,61h,00h,2Bh,00h,0Ah,0C2h,2Ch,31h,00
defb 44h,00h,0D8h,0DBh,00h,31h,00h,3Ah,00h,01h,14h,42h,00h,3Eh,00h,50h
defb 61h,00h,2Bh,00h,0Ah,42h,00h,48h,00h,0CAh,5Dh,4Fh,00h,5Fh,00h,01h
defb 0E4h,0C8h,00h,4Fh,00h,57h,00h,01h,42h,00h,5Bh,00h,42h,5Dh,48h,00h
defb 02h,00h,90h,6Dh,80h,69h,00h,0Bh
defw     ques
defb 10h,82h,00h,0Ch,70h,0B2h
defb 00h,73h,00h,0Dh,0CAh,2Eh,7Ah,00h,80h,00h,01h,70h,0BDh,00h,73h,00h
defb 0Eh,02h,00h,90h,92h,00h,88h,00h,0Fh,90h,9Dh,00h,8Eh,00h,10h,10h
defb 0A8h,00h,11h,42h,2Ah,96h,00h,42h,2Eh,9Ah,00h,20h,0BDh,00h,60h,0B2h
defb 00h,0A6h,00h,42h,2Eh,0A6h,00h,02h,2Ah,42h,2Ah,0ACh,00h,42h,2Eh,0B0h
defb 00h,02h,2Ah,0CCh,30h,01h,0B2h,00h,0BBh,00h,08h,01h,02h,00h,0CCh,30h
defb 01h,0BDh,00h,0C6h,00h,03h,01h,02h,00h,0C0h,0E4h,00h,0CFh,00h,0D8h,00h
defb 0CAh,06h,0CFh,00h,0D6h,00h,01h,02h,00h,00h,0F6h,00h,60h,0E2h,00h,0E0h
defb 00h,02h,3Ah,02h,05h,82h,44h,0E8h,00h,82h,47h,0ECh,00h,82h,4Eh,0F0h
defb 00h,82h,50h,0F4h,00h,02h,54h,82h,42h,0FAh,00h,82h,45h,0FEh,00h,82h
defb 46h,02h,01h,82h,48h,06h,01h,82h,49h,0Ah,01h,82h,4Ch,0Eh,01h,82h
defb 4Eh,12h,01h,82h,4Fh,16h,01h,82h,51h,1Ah,01h,82h,52h,1Eh,01h,82h
defb 53h,22h,01h,82h,55h,26h,01h,82h,56h,2Ah,01h,82h,57h,2Eh,01h,02h
defb 5Ah,82h,05h,34h,01h,82h,06h,38h,01h,02h,07h

        end
```

End Listing Six

Towards a More Writeable Forth Syntax*

by Harvey Glass

While Forth is a powerful and efficient language, it may also be justly criticized for creating unnecessary programmer obstacles. Particularly objectionable are the stack operators which tend to obscure the original solution, increase programmer error, and result in code which is often virtually unreadable.

We will describe a set of Forth extensions designed to reduce programming effort. These extensions provide the following features:

(1) A programming style that requires no explicit parameter stack operations,
(2) A straightforward notation for defining recursive functions,
(3) The handling of forward references, and
(4) A new construct for defining higher order functions that is similar to CREATE DOES> but easier to use.

The principal goal in designing a programming language should be to reduce the effort required to translate a problem solution into a machine language. Our specific goal with the extensions to Forth has been to preserve the power and flexibility of the language while providing a simpler and more readable notation. The resulting code is not unlike a postfix dialect of Pascal or Algol. We will discuss the syntax of the extensions and their implementations.

Forth programmers tend to be impatient with attacks on the syntax of the language, since the advantages of Forth seem clearly to outweigh any minor inconveniences. Yet Forth does have shortcomings, as some critics have repeatedly pointed out.[1,2] The most common complaint is the lack of readability. We will demonstrate extensions to the language which provide a "better" syntax, one which is entirely free of stack operations. By "better" we mean both more writable and more readable.

Harvey Glass, University of So. Florida, College of Engineering, Department of Computer Science and Engineering, Tampa, Florida 33620.

Copyright © 1983 by Harvey Glass. General permission is granted to republish, but not for profit, all or part of this material, provided that Harvey Glass's copyright notice is given and reference is made to the original publication, its date of issue, and the fact that reprinting privileges were granted by Harvey Glass.*

**Reprinted from the* Proceedings of the 1983 Rochester Forth Applications Conference *with permission of the author and the publisher, the Institute for Applied Forth Research, Inc.*

An Example With Recursion

Consider this example of the Ackermann function described by Peterson.[3] The function ack(i, j) is recursive and is given by

```
j + 1                          if i = 0
ack(i - 1, 1)                  if j = 0
ack(j -1, ( ack(i, j - 1) )    otherwise.
```

In Forth the function ack is defined recursively as shown in Figure 1 (page 82). The definition is perhaps not unreadable, but neither is it clearly related to the original formulation. Suppose we restate the original function definition in reverse Polish:

```
if I = 0       J 1 +
if J = 0       I 1 - 1 ACK
otherwise      J 1 - I J 1 - ACK ACK.
```

With the extensions, the Forth colon definition is rewritten with a new notation where the parameters i and j are declared. The resulting program definition is very close to the problem definition – in reverse Polish. Note that the words <DEFINE and >> in Figure 2 (page 82) play the roles of : and ; respectively.

Notice that the new programming definition is virtually identical to the restated function definition. We suggest that the new syntax is not only more readable but is more writable. For this example and similar problems, the notation greatly reduces the time to resolve programming errors. Our goal is simply a syntax which relates directly to the statement of the solution. This makes the programmer's job a great deal easier. We can go even further in providing a more readable definition. If we create a new vocabulary within which ((left parenthesis),) (right parenthesis), and , (comma) are redefined as immediate no-operations, the definitions of ACK may be rewritten as demonstrated in Figure 3 (page 82).

In a later section we sketch implementation of the extensions. Note that:

(a) The new definitions may be mixed freely with Forth colon definitions.
(b) Declared variables (I and J above) can be referenced only inside the definition. They occupy no permanent dictionary space.

A Simple Example

The new approach is useful for nonrecursive functions as well. Consider the comparatively simple problem of calculating the value a*c – b/c given the values of a, b, and c. The Forth glossary entry is

```
VAL   A B C --- A*C - B/C
```

In Forth we write

```
: VAL ROT OVER * ROT ROT / - ;
```

We suggest that the relationship between the expression A*C – B/C and the expression ROT OVER * ROT ROT / – is not obvious. It is possible, however, to make the programmer's job easier, reduce the chances of error, and provide a more readable definition.

The expression A * C – B / C is written in reverse Polish as

(a) A C * B C / –

The corresponding definition in the new notation is

```
<DECLARE VAR A VAR B VAR C
<DEFINE VAL
(b)    ( returns ) A C * B C / - >>
```

Note that the expressions (a) and (b) above are identical. We merely rewrite our solution as our programming definition.

Implementation and Efficiency

The extensions require that a stack called the lambda stack be added to the Forth system. Values corresponding to the declared variables are removed from the parameter stack at execution and pushed onto the lambda stack. A declared variable used in a definition will retrieve the corresponding value from the lambda stack at execution and push it onto the parameter stack.

The names of declared variables are allocated space in a portion of the dictionary available only during compilation. This space is reused for subsequent definitions so that declared variables require no permanent dictionary space.

Preliminary experiments with the extensions indicate marginal increases in execution time of 20 to 30 percent and even less impact upon memory requirements. On the other hand, programmer

time for coding and debugging will typically be halved.

Temporary Values

The extensions include a feature that handles intermediate or temporary values. It eases the bookkeeping burden of a Forth programmer who must otherwise keep cuff records of values on the parameter stack and the return stack. Suppose we extend the computation of the previous example. We compute the value

NEWVAL
A B C---$(A*C-B/C)^2+(A*C-B/C)-C^2$

Since we need to compute the value of A*C-B/C only once, we introduce the idea of a temporary value and rewrite the expression in reverse Polish as

A C * B C / - TEMP LET
(i.e. : let temp = [a*c-b/c])

and then calculate

TEMP TEMP * TEMP + C C * -

In Forth NEWVAL is defined as shown in Figure 4 (page 82). The stack operations are only distantly related to the original problem. Contrast this with the example in Figure 5 (page 83). The word LOCAL defines temporary storage local to the new definition. No permanent dictionary storage is needed for the name TEMP.

Forward References

One of Forth's structural requirements is that all words must be defined before they are used. While this is not generally a problem, it can be constraining. Often we rewrite a definition, use a word not yet defined, and then discover that correcting the difficulty requires a messy reordering of the load sequence.

The word FORWARD relieves the problem in a simple way. To use a word not yet defined we write

FORWARD < name of new word >

The word FORWARD builds a definition which looks like

: < name of new word >
ERROR-ROUTINE ;

where execution of ERROR-ROUTINE displays an error message. Later in the load, a definition of the form

<DECLARE . . . <DEFINE
< name of new word >. . . >>

replaces the address of ERROR-ROUTINE in the colon definition created by FORWARD with the address of the new word.

The process is simple, and the syntax convenient.

A New High Level Defining Construct

The CREATE DOES> facility in Forth allows the definition of words which are themselves defining words.[4,5] CREATE DOES> is a powerful capability found in few computer programming languages, but because the construct is not simple to use, it is often avoided by less experienced programmers. We will describe a syntax for the definition of defining words which provides a capability similar to CREATE DOES> but with a simpler notation. Consider the Forth definition of the defining word MATRIX.

MATRIX, given the parameters M and N, allocates an M * N byte array and defines a new word which requires the parameters I and J. The new word, when executed, insures that I and J are in the range

$$1 \leqslant I \leqslant M \text{ and } 1 \leqslant J \leqslant N$$

If they are not, an error message is displayed. Otherwise the location of the I, Jth byte in the array is returned.

The definition of MATRIX in Forth is shown in Figure 6 (page 83). Contrast this with the new definition in Figure 7 (page 83):

> Note that in this definition, the word CALLOCATE reserves bytes (ALLOCATE reserves words). The word ADDRESS will return the address of the first reserved location. Also note that the location of the I, Jth element is given by
>
> ADDRESS+N*(I-1)+(J-1)
>
> which in reverse Polish is
>
> ADDRESS N I 1 - * + J 1 - +

The new definition is displayed in Figure 7 (page 83).

Summary

While technology, each year, provides more powerful hardware at less cost, our software problems seem to defy solution. Forth provides unique capabilities, but if Forth is to be accepted by large numbers of programmers it will be because Forth makes programming easier. Our goal in developing the extensions has been to add to Forth's capability by providing a notation that more directly relates a programming solution to the original problem. We feel that the extensions are a step towards this goal. The impact upon machine efficiency is marginal while programmer productivity is considerably enhanced.

References

[1] Paul Freiberger, et al., August 11, 1982, "In Focus," *InfoWorld*, pp. 15-36.

[2] R.G. Loeliger, September 1981, "Sallying Forth to Battle," *Dr. Dobb's Journal*, 59, pp. 25-26.

[3] Joel V. Petersen, 1981, "Recursion and the Ackermann Function," *Forth Dimensions III, 3*, pp. 89-90.

[4] Michael Perry, 1982 "Charles Moore's BASIC Compiler Revisited," *Forth Dimensions III, 6*, pp. 175-178.

[5] William F. Ragsdale, 1981, "A New Syntax for Defining Defining Words," *Forth Dimensions II, 5*, pp. 121-128.

Further Reading

Leo Brodie, 1981, "Starting Forth," Prentice Hall.

Bob Gotsch, 1982, "The Art of Recursion," *Forth Dimensions IV, 2*, pp. 24-26.

FIG. 1

```
: ACK       ( i j --- ack[i, j] )
  DUP Ø= IF SWAP DROP 1+
         ELSE DUP Ø= IF DROP 1 - 1 MYSELF
                     ELSE OVER 1 - ROT ROT
                          1 - MYSELF MYSELF
                     THEN
         THEN   ( returns ack[i, j] )     ;
```

FIG. 2

```
<DECLARE VAR I VAR J
<DEFINE ACK
  I Ø= IF
                        J 1 +
       ELSE J Ø= IF     I 1 - 1 ACK     .
                ELSE    J 1 - I J 1 - ACK ACK
                THEN
       THEN  ( return ack[i, j] )  >>
```

FIG. 3

```
<DECLARE VAR I VAR J
<DEFINE ACK
  I Ø= IF             J 1 +
       ELSE J Ø= IF ( I 1 - , 1 ) ACK
                ELSE ( ( J 1 - , ( I , J 1 - ) ACK ) ACK
                THEN
       THEN           >>
```

FIGURE 4

```
: NEWVAL     ( a b c --- [a*c-b*c]**2 + [a*c-b/c] - c**2 )
     DUP >R        ( save c )
     ROT OVER *    ( a*c )
     ROT ROT /     ( b/c )
     -             ( temp = [a*c-b/c] )
     DUP DUP *     ( temp*temp )
     +             ( temp*temp+temp )
     R> DUP * - ;  ( temp*temp+temp-c*c )
```

FIGURE 5

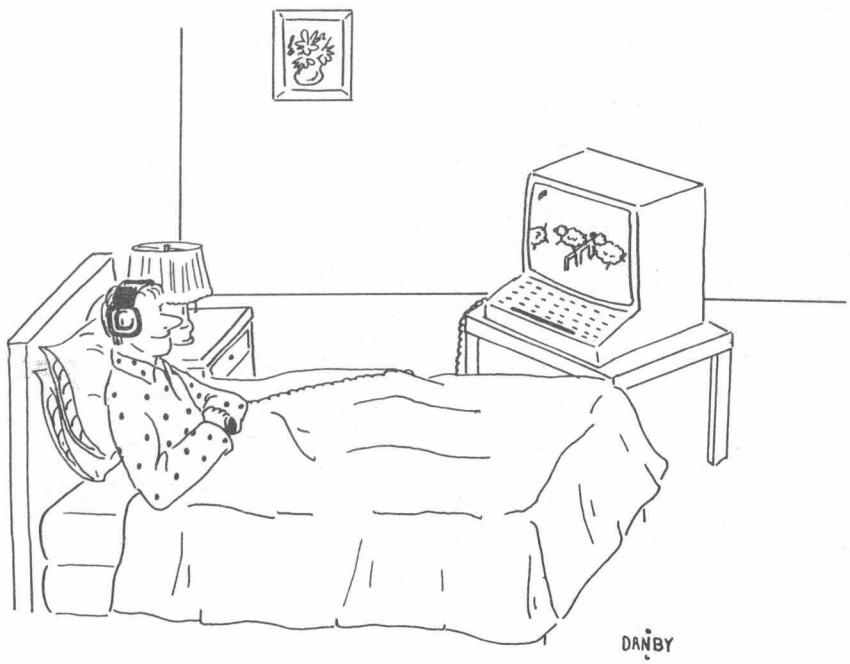
```
<DECLARE VAR A VAR B VAR C LOCAL TEMP
 <DEFINE NEWVAL
     A C * B C / -  TEMP LET  ( set temp to a*c-b/c )
   ( returns )   TEMP TEMP *  TEMP + C C * -    >>
```

FIGURE 6

```
: MATRIX
  CREATE 2DUP , , * ALLOT
  DOES>
        ROT DUP 1 < >R      ( test i < 1 )
        ROT DUP 1 < >R      ( test j < 1 )
        ROT DUP @           ( get value of n )
        ROT DUP ROT > >R    ( test j > n )
        ROT ROT DUP 2+ @    ( get value of m )
        ROT DUP ROT >       ( test i > m )
        R> OR R> OR R> OR   ( bounds test )
        IF  2DROP DROP
            ." BOUNDS ERROR "
        ELSE  1 - ROT 1 - SWAP    ( calculate byte location )
              ROT DUP @ ROT *     ( = base + n*[i-1] )
              ROT + 4 + +         ( + [j-1] + 4  )
        THEN    ;
```

FIGURE 7

```
<DECLARE VAR M VAR N          ( parameters for the defining word )
 DECLARE VAR I VAR J          ( parameters for the new word )
<DEFINER MATRIX               ( definition time action )
        M N * CALLOCATE       ( reserve m*n bytes )
<FORM                         ( this is what the word does )
          I 1 < J 1 < OR        ( test i < 1 and j < 1 )
          I M > J N > OR OR     ( test i > m and j > n )
   IF     ." BOUNDS ERROR "     ( bounds error if they are )
   ELSE                         ( else return byte location = )
    ADDRESS N I 1 - * + J 1 - + ( ADDRESS + n*[i-1] +[j-1] )
   THEN       >>>
```



Simple Graphics for Printer

by Wil Baden

Wil Baden, 339 Princeton Drive, Costa Mesa, California 92626.

Marek W. Michalski, in "A Simple Vector Generation Algorithm" (*DDJ* No. 74, December 1982, pp. 58f), reinvents (almost) the classic algorithm for plotting straight lines. This was independently discovered about 20 years ago by a number of people, and was published as "Algorithm for Computer Control of Digital Plotter," written by J. E. Bresenham in the *IBM System Journal*, 4(1) 1965, pp. 25-30.

The algorithm is, of course, intended to be implemented in machine language, and the arithmetic calculations are minimal – using just addition, subtraction, and perhaps left shift (to multiply by 2). It is important to avoid using time-consuming multiplication and division. Also, the actual inner loop is very simple.

Anyone who is interested in algorithms or graphics should know about the Bresenham algorithm. It not only does the job, it does it well. The Michalski algorithm requires more computation, does not always make the best choice for its first move, and plots two points when it should plot only one in the degenerative case of zero distance.

A Pascal program is provided which uses the Bresenham algorithm for *very*

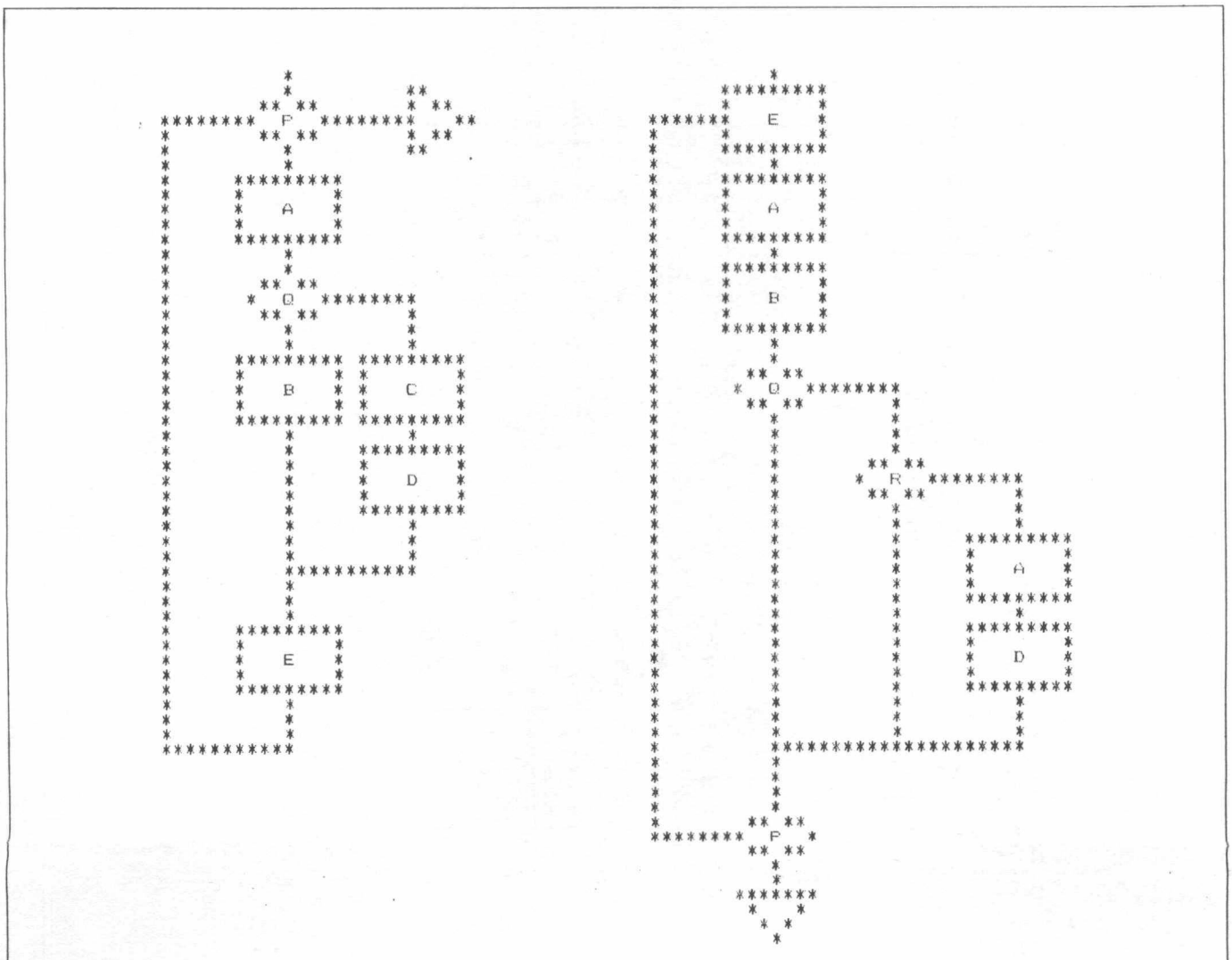


Figure 1.
Flowchart diagrams depicting the inner loops of the Bresenham and Michalski algorithms.

low resolution graphics on your terminal or line printer (though the algorithm can also be the basis for much higher resolution graphics routines). Procedures are provided for move absolute, move relative, line absolute, line relative, draw lozenges and oblongs, and for drawing flowcharts of the inner loop of the Bresenham and Michalski algorithms.

The Pascal code (see Listing, below) is in standard Pascal, and should thus work on most Pascal compilers. The letters in the flowchart boxes in Figure 1 (page 86) may be found in the comments of the Pascal listing (page 89, lines 87-111). As the flowcharts illustrate, the Bresenham algorithm has the simpler structure. DDJ

(Listing begins below)

Simple Graphics

Listing

```
 1  program plotting (input,output) ;       { simple graphics for printer }
 2  const
 3      XMAX    = 80 ;                                  { horizontal rasters }
 4      YMAX    = 88 ;                                  { vertical rasters }
 5      MAXSAVE = 20 ;                                  { stack size }
 6  type
 7      xraster = 1..XMAX ;
 8      yraster = 1..YMAX ;
 9  var                                                 { global variables }
10      pixel   : array[xraster,yraster] of char ;      { the graphic window }
11      x       : xraster ;                             { current position }
12      y       : yraster ;
13      s       : 1..MAXSAVE ;                          { stack pointer }
14      xsave   : array[1..MAXSAVE] of xraster ;        { stack }
15      ysave   : array[1..MAXSAVE] of yraster ;
16  function sgn (x : integer) : integer ;              { signum of x }
17  begin
18      if      (x < 0) then sgn := -1
19      else if (x = 0) then sgn := 0
20      else    {x > 0}      sgn := 1
21      end ;
22  function abs (x : integer) : integer ;              { absolute value of x }
23  begin
24      if (x < 0) then abs := -x
25      else            abs := x
26      end ;
27  procedure mabs (x1 : xraster ; y1 : yraster ) ;     { move absolute }
28  begin                          { beware -- lacks range check }
29      x := x1 ;
30      y := y1
31      end ;
32  procedure mrel (dx : integer ; dy : integer ) ;     { move relative }
33  begin
34      mabs (x+dx, y+dy)
35      end ;
36  procedure setup ;                                   { initiation }
37  var x1 : xraster ;  y1 : yraster ;
38  begin
39      for y1 := 1 to YMAX do
40          for x1 := 1 to XMAX do
41              pixel[x1, y1] := ' ' ;
42      x := 1 ;
```

(Continued on next page)

Simple Graphics (Listing continued, text begins on page 86)

Listing

```
43      y := 1 ;
44      s := 0
45      end ;
46  procedure cleanup ;                                  { termination }
47  var x1 : xraster ;  y1 : yraster ;
48  begin
49      writeln ;
50      for y1 := 1 to YMAX do begin
51          for x1 := 1 to XMAX do
52              write(pixel[x1,y1]) ;
53          writeln ;
54          end
55      end ;
56  procedure fork ;                                     { save current position }
57  begin                           { beware -- lacks overflow check }
58      s        := s + 1 ;
59      xsave[s] := x ;
60      ysave[s] := y
61      end ;
62  procedure back ;                                     { revert to old position }
63  begin                           { beware -- lacks underflow check }
64      mabs (xsave[s], ysave[s]) ;
65      s := s -1
66      end ;
67  procedure cabs (x1 : xraster ; y1 : yraster ; marker : char) ;
68  begin
69      mabs (x1, y1) ;
70      pixel[x,y] := marker
71      end ;
72  procedure labs (xf : xraster ; yf : yraster) ;   { line absolute }
73  var sx, sy : -1..1 ;            { increment to next pixel }
74      dx, dy : integer ;          { projection of vector on axes }
75      f      : integer ;          { direction finder }
76  begin
77      cabs (x,y,'*') ;
78      dy := abs(yf-y) ;           { vertical movement }
79      sy := sgn(yf-y) ;           { vertical direction }
80      dx := abs(xf-x) ;           { horizontal movement }
81      sx := sgn(xf-x) ;           { horizontal direction }
82      if (dx < dy) then begin
83          { Bresenham algorithm for prolate motion }
84          dx := 2 * dx ;          { short adjustment }
85          f  := dx - dy ;         { initial direction }
86          dy := f - dy ;          { long adjustment }
87          while (y <> yf) do begin                          { P }
88              y := y + sy ;       { move on long axis }     { A }
89              if (f < 0) then                               { Q }
90                  f := f + dx     { add short distance }    { B }
91              else begin
92                  x := x + sx ; { move on diagonal }        { C }
93                  f := f + dy     { add adjustment }        { D }
94                  end ;
```

```
 95              cabs (x, y, '*')  { next point }                { E }
 96              end
 97          end
 98      else {dx >= dy} begin
 99          { Bresenham algorithm for oblate motion }
100          dy := 2 * dy ;          { short adjustment }
101          f  := dy - dx ;         { initial direction }
102          dx := f - dx ;          { long adjustment }
103          while (x <> xf) do begin                              { P }
104              x := x + sx ;       { move on long axis }         { A }
105              if (f < 0) then                                   { Q }
106                  f := f + dy     { add short distance }        { B }
107              else begin
108                  y := y + sy ; { move on diagonal }            { C }
109                  f := f + dx   { add adjustment }              { D }
110                  end ;
111              cabs (x, y, '*')  { next point }                  { E }
112              end
113          end
114      end ;
115  procedure lrel (dx : integer; dy : integer);     { line relative }
116  begin
117      labs (x+dx, y+dy)
118      end ;
119  procedure diamond (marker : char) ;               { draw a lozenge }
120  begin                          { enter at top }
121      labs (x,  y+1) ;
122      labs (x+3,y+2) ;
123      labs (x-3,y+2) ;
124      labs (x-3,y-2) ;
125      labs (x+3,y-2) ;
126      cabs (x,  y+2, marker) ;
127      mabs (x,  y+2) ;
128      labs (x,  y+1)
129      end;
130  procedure box (marker : char) ;                   { draw an oblong }
131  begin                          { enter at top }
132      labs (x,  y+1) ;
133      labs (x+4,y) ;
134      labs (x,  y+4) ;
135      labs (x-8,y) ;
136      labs (x,  y-4) ;
137      labs (x+4,y) ;
138      cabs (x,  y+2, marker) ;
139      mabs (x,  y+2) ;
140      labs (x,  y+1)
141      end ;
142  procedure bresenham ;                             { Bresenham algorithm }
143  begin
144      fork ; mrel (4,3) ; lrel (6,0) ;
145          lrel (0,-2) ; lrel (5,2) ; lrel (-5,2) ; lrel (0,-2) ; back ;
146      diamond('P') ;  box('A') ;
147      fork ; mrel (4,3) ; lrel (6,0) ; lrel (0 ,3) ;  box('C') ;
```

(Continued on next page)

Simple Graphics (Listing continued, text begins on page 86)

Listing

```
148          box('D') ; lrel (0,3) ; lrel (-10,0) ; back ;
149      diamond('Q') ; box('B') ; lrel (0,12) ; box('E') ;
150      lrel (0, 3) ;  lrel (-10,0) ; lrel (0,-42) ; lrel (6,0) ;
151      end ;
152  procedure michalski ;                          { Michalski algorithm }
153  begin
154      box('E') ; box('A') ; box('B') ;
155      fork ; mrel (4,3) ; lrel (6,0) ; lrel (0,3) ;
156          fork ; diamond('R') ; lrel (0,15) ; back ;
157          mrel (4,3) ; lrel (6,0) ; lrel (0,3) ;
158          box('C') ; box('D') ; lrel (0,3) ; lrel (-20,0) ; back ;
159      diamond('Q') ; lrel (0,24) ; diamond('P') ;
160      fork ; mrel (-4,-3) ; lrel (-6,0) ; lrel (0,-48) ; lrel (6,0) ; back ;
161      lrel (0,1) ; lrel (-3,0) ; lrel(3,3) ; lrel(3,-3) ; lrel(-3,0) ;
162      end ;
163  begin                                          { main }
164      setup ; mabs (11,1) ; bresenham ;  mabs (51,1) ; michalski ; cleanup ;
165      end .
```

End Listing

(Continued from page 661)

```
task_1_sp                       B20         2848        0x01
task_2                          B20         2848        0x01
task_2_sp                       C44         3140        0x01
task_3                          C44         3140        0x01
task_3_sp                       D6C         3436        0x01

tenth_constant                 C350        50000        0x01
trap_0                           80          128        0x01
user_sr_mask                      0            0        0x01
wait                            958         2392        0x01
was_super                       9D4         2516        0x01
```

End Listing

WP6502 V1.3a

Company: Dwo Quong Fok Lok Sow, 548 Broadway, Suite 4F, New York, NY 10012

Computer: M/A-COM OSI (Ohio Scientific), versions OS65U and OS65D

Price: $300.00 for V.OS65U and $250.00 for V.OS65D

Circle Reader Service No. 125

Reviewed by James Criscione

Those of us who actively use OSI systems in our everyday lives constantly envy our associates who have products such as WordStar, which perform their word processing functions. Over the past several years, OSI has turned out a number of word processors (the WP-X series) that are simply redoings and patchings of their early editor/assembler. This has left the OSI user in a wasteland into which WP6502 by DQFLS fits in sufficiently.

As word processors go, it would best be described as "middle of the road." It is very good at handling text for insertions, deletions, additions, and displaying this in a variety of proportional/right justified modes, depending on the type of printer. True proportional spacing can be accomplished only on parallel printers. Serial printers use either microproportional spacing (varying the spacing between words in small increments) or a simple justification mode based on whole space increments between words depending on the capabilities of the printer.

The WP6502 system is menu-driven so that it is very easy for the novice user to begin word processing. The system is booted and an Install? message is displayed. This gives the user the opportunity to reconfigure the system to the type of CRT and printer currently in use. It is only necessary to do this one time for each system as these parameters are saved to disk.

If the install option is not selected, the menu displays: Type, View, Blk view, G.Edit, L.Edit, Move, Delete, Zap, Send, Receive, and Clerk. Type is the mode selected to input text to the word processor. Text is displayed as entered, and it is not necessary to enter "RETURN" at the end of the line because the system does this automatically. View is selected if the operator wishes to see the display of text either on the CRT or the printer. The system also allows "Blocks" of text that may be predefined and inserted at any point in work in progress simply by entering #Bnn where nn is the block of text to select. This is useful in form letters, or as a boilerplate for legal documents. One drawback is that the text must be contained in memory, so you are limited to the size of the blocks that may be accessed at any one time.

The system also allows for both global and line edits in a fairly standard fashion. Text may also be moved from one location to another, and the user has the option of either retaining the existing text and the moved text or having the system automatically delete the old text. The delete command is useful for deleting large blocks of text although most text will probably be deleted in the line edit mode. Zap is to clear the memory prior to entering text.

The Send and Receive options are supposed to allow the user to transfer text from one system to another via common printer ports. This portion does not work as specified and we have not been able to get adequate explanation from DQFLS despite numerous calls, most of which went unreturned. I suspect from the method in which the few calls were returned that DQFLS is a part time software house and the programmers have other employment; for in no case was anyone ever available directly who could answer any questions regarding the software.

The CLERK command on the menu allows the user to manipulate the actual disk files (save, load, create new files, etc.). This is an improvement on DQFLS's prior offerings in which this had to be done outside of the menu.

In summary, WP6502 is an adequate word processing system for OSI mini's. It does not have many of the features that a more sophisticated system might offer, but, for the simple manipulation of text, does reasonably well. The programs seem to run fairly well so I think the difficulty with software support will not be really crucial.

DATA CAPTURE/pc

Company: Southeastern Software, 7743 Briarwood Drive, New Orleans, LA 70128

Computer: IBM PC with 128K Memory

Price: $120

Circle Reader Service No. 127

Reviewed by Chuck Crayne

DATA CAPTURE/pc is a terminal emulation program that supports the asynchronous adapter card for the IBM personal computer. Like the other communications packages on the market, this program will allow the IBM PC user to communicate with host computers, public access data networks, and other personal computers. The unique feature of this particular program is that it contains a text buffer that can capture about 750 lines of 40 characters. When buffer capture is in effect, all input and outpu are stored into the buffer at the same time that they appear on the screen.

When I was a HAM TTY operator, it was considered good form to pre-type (on paper tape) outgoing messages before going on the air. This practice not only saved a lot of time for the other operators on the net, but avoided exposing my own fumble-fingered typing skills to the world. On today's computer nets, an additional factor applies – the connect charge.

The capture buffer makes the process of pre-formatting messages a lot easier than using punched paper tape. After the program is initialized, but before it's logged onto the net, you can type your messages onto the screen. They will be captured in the buffer. Then, when you are signed into the right section of the net, you can easily cause any or all of the messages to be sent.

A similar function could be done with programs that support file uploads, but it would require that the files be previously created with some sort of text editor. Even then, to match the versatility of DATA CAPTURE, each message would have to be in a separate file.

If the capture buffer overflows, it will automatically be written to disk. Alternatively, the user can choose to capture directly to disk, send from disk, or do file maintenance functions, all without leaving the communications environment.

The capture buffer has a price, however. The current version of the program requires a 128K system. Without the buffer, it could either run on a smaller system or contain additional features. There is no dialing directory, for example. Nor is there any provision (such as the XMODEM protocol) for transferring non-text files.

The functions that do exist, however, are easy to use. The beginner or occasional user will be pleased with the menu-driven approach to reaching the desired function. The expert user can bypass the menu and

reach any function directly from the communication screen. The documentation is above average, containing both an indexed reference manual and a 17-page tutorial.

The current version of DATA CAPTURE/pc is DOS 2.0 compatible. Southeastern Software will upgrade previous versions for $5.00 with return of the original master diskette. Any diskette whose serial number starts with the letter "B" is the DOS 2.0 version. Serial numbers starting with "A," or no letter at all, should get the new version if they plan to use DOS 2.0.

DATA CAPTURE/pc gets good marks for documentation and ease of use. All data communications packages for the IBM PC must, however, be compared to Andrew Fluegelman's PC-TALK III. PC-TALK has become extremely popular due to the FREEWARE approach in which each user of the program is encouraged to give away copies. The recipient of each copy is encouraged, but not required, to send $35 to the program author.

To compete against this approach, a program selling at $120 must offer something of significant value that is not available in PC-TALK. The unique feature that DATA CAPTURE/pc offers is the capture buffer. If you are paying significant amounts in access charges for 300 or 1200 baud communications while you type at 30 or less, this program could be a good investment.

/HELP

Company: Southeastern Software, 7743 Briarwood Drive, New Orleans, LA 70128

Computer: IBM PC with one drive and 128K, or COMPAQ with one drive and 192K

Price: $39.50

Circle Reader Service No. 129

Reviewed by Diane Crayne

Every beginning BASIC programmer needs /HELP! This is one of the nicest little programming aids to come along in quite a while. In fact, it is so nice and simple that I wonder why someone didn't think of it a long time ago.

/HELP is a memory resident program that you can call up to give you a short description of practically any BASIC command. And you can do this right in the middle of coding a program, without losing track of where you are, or interfering with what you're doing. You car also use /HELP to describe the most common DOS commands.

This nifty little memory aid runs on the IBM PC and the COMPAQ. Its only drawback is the memory requirement of 42K; it needs 128K to run on the PC, and 192K on the COMPAQ. Unfortunately, this puts it beyond the reach of the people who will probably have the most need for it – the beginning programmers who may have a 64K system.

I tested /HELP on a 128K IBM PC with two dual drives. The only documentation I had was the little nine-page booklet that comes with the software. That was all I needed. Installation is a true snap. Format the disk you're going to use for your next BASIC program and put it in drive B:. Put the /HELP disk in drive A:. Log on to drive A:. Type MAKE I if you're using a PC, and MAKE C if you're using a COMPAQ. /HELP copies onto the target disk and installs itself into memory when you log onto B: and type HELP. That's all there is to it. The booklet also has installation instructions for people with single-drive systems.

Once /HELP is installed, all you have to do is type / and a friendly little message appears at the top of the screen: "What would you like /HELP with?" You can either press the enter key and go back to whatever you were doing (/HELP blanks the top ten lines of the screen for its messages), type ? to see a list of the commands you can get help with, or type in the name of a command and get a quick refresher on how it works.

Naturally the descriptions aren't exhaustive – you can only do so much in ten lines – but they show one or two examples and that's usually enough to remind you about the syntax. If what you're doing is too arcane for /HELP to cover, you can always refer to the proper page in the IBM PC BASIC manual, which is neatly printed in brackets at the top right of the /HELP area.

If you don't like using the / key to invoke /HELP, you can change the call character to anything you want. All you have to do is hit the slash key a second time when the prompt appears, and you'll be prompted for a new character. The manual suggests that you use a printable character and not a control or funtion key. Being of a contrary nature, I selected the F1 function key. /HELP seemed to run fine, but the prompt line didn't completely display on the screen. So, I bowed to the wiser heads behind the product and chose the = key instead.

I had a particular reason for picking = as the call character. It is one of the most frequently used characters in BASIC programming. I wanted to know if it would mess up either /HELP or my program; it didn't. You can't call /HELP if you have entered a line number and are in the middle of editing a BASIC statement. Once the command description has been displayed, you can type the access character again and request another description. This eliminates having to toggle back and forth between the program screen and the /HELP screen when you have more than one query.

/HELP can be a very useful tool for the beginning programmer. Its descriptions are concise; its examples are easy to understand; and its $39.50 price tag is quite attractive. It's a tool with a limited life-span (since the programmers will outgrow it); but for the time it's being used, it will save hours of frustrated searching through the manual.

Displaying the BASIC manual page numbers is a nice feature, but the numbers only match version 2.0. If you have an earlier version, or if IBM releases a newer one, the numbers will be out of synch.

/HELP's main problem is the 128K memory requirement. There are evidently a few quirks lurking in the bushes too, since the manual advises against having /HELP in memory when you're running other programs. The results, they say, "can be unpredictable." When you use the program the way it's intended, the results are quite predictable – and very good!

SuperWriter

Company: Sorcim Corporation, 2310 Lundy Avenue, San Jose, CA 94131

Computer: IBM PC

Price: $295

Circle Reader Service No. 131

Reviewed by Ronald G. Parsons

I have used many word processors during the past six or seven years – some on large mainframes, some I've written myself, some stand-alone units, and some on personal computers. SuperWriter beats them all in one very important category – ease in learning to use it. Just a few hours after receiving this copy for review, I had converted a large document from WordStar and was modifying it with SuperWriter.

It is only natural (and reasonable) to compare a personal computer word processor to WordStar as it is one of the current best sellers. My main experience with WordStar is with their CP/M version using a memory-mapped video board, but that version performed better than the versions I've used on the IBM PC (I understand the new version of WordStar for the IBM PC does correct some of the extreme slowness of screen updates).

SuperWriter has the capabilities of WordStar and its MailMerge and SpellStar companions in one package at about a third of the suggested retail price. I found no significant function available from WordStar and similar packages that was not present in SuperWriter. The excellent help screen facility available in SuperWriter and the speed of screen updates make the package a very good value.

I have had three occasions to call

Sorcim's customer service department with questions on SuperCalc and SuperWriter and the response has always been swift and accurate (even one day when their phones were out of order and *they* called *me* back). Micropro, on the other hand, took over six months to respond to a problem with SpellStar and, after repeated calls and letters from me, finally responded that nothing could be done – "that's the way it is." Sorcim's manuals are clear, easy to use, and professionaly done. Micropro's on the other hand... (I've heard their new ones have finally been redone). I found that I needed to use only the SuperWriter reference card and the help screens with occasional references to the index.

This is not to say that SuperWriter does not have its faults. Like WordStar, there is no "undo" command to reverse an erroneous deletion. Keys on the IBM PC's keyboard, such as PgUp, PgDn, and End are not used for their labeled functions (I'll describe a cure for this problem using PCDOS 2.0 a little later). The printer I use has a 2000-character buffer and the printer driver times-out waiting for the buffer to empty.

One of my favorite features of WordStar is absent – page breaks in SuperWriter are not visible on the screen while you are editing. Provision is available to prevent widows and orphans (those single lines at the beginning and end of paragraphs that end up respectively on the bottoms and tops of pages) but I like to be able to see where the page ends.

I like several features on SuperWriter much better than on WordStar. In SuperWriter, for example, as a paragraph is modified, the line ends are readjusted in real time rather than waiting for the user to request reforming of the paragraph. When the cursor is moved vertically in WordStar, it jumps to the left if it passes a short or zero length line.The SuperWriter cursor stays in a single column. The visible indications for bold and underline are much more mnemonic than the cryptic control symbols on WordStar. I get the feeling that I am using three separate packages when using WordStar/MailMerge/SpellStar, whereas the SuperWriter features are integrated much better as a single package.

The form letter generation capabilities are very similar. However, SuperWriter has the capability of reading fixed-format data files. The procedural capabilities with IF logic are quite good. The SuperWriter spelling checker is much faster than SpellStar. On two identical files of around 3400 words, SpellStar took 127 seconds to complete the check while SuperWriter took only 23 seconds. SpellStar was running with diskettes while SuperWriter was on a fixed disk on a PC XT, which could explain some of the difference, but I'm sure that much of the time difference is real. Each package has a 20,000+ word dictionary. The SuperWriter spelling checker flagged 105 "words" while SpellStar flagged 88. Both dictionaries are weak on plurals and words with suffixes and prefixes. SpellStar often flags words with apostrophes while SuperWriter flagged many with hyphens.

SuperWriter lets you review the mismatched word list so the "correct" words can be eliminated before marking them in the text. This is very useful in long documents with a large amount of jargon. Adding words to the SuperWriter dictionary is very easy to do. SpellStar, on the other hand, marks all or none of its flagged words in the body of the text. Words in this review that were flagged by the two spelling checkers (eliminating proper nouns, abbreviations, etc.) are listed below:

Flagged by both: configures cure directives mainframes menus mnemonic nouns orphans overridden overrides phones plurals redefines redefinition redefinitions redone reentered reforming sellers widows

SuperWriter flagged readjusted verification (plus most hyphenated words)

SpellStar that's author's mismatched Sorcim Micropro ASCII

Isn't it interesting that SpellStar marked Sorcim *and* Micropro as misspelled, but SuperWriter marked neither!

A Typical Session

The system is organized around a series of full screen menus. The main menu presents a choice of EDITing a document, QUITing to the operating system, PRINTing a document, CHECKing spelling in a document, getting a list of the DISK directory, or executing a set of provided UTILITIES. Choosing EDIT (the default choice – others are made by moving the cursor with the space bar and pressing enter) causes a filename to be requested. If it is to be a new file, a document history is started (author's name is requested) and a blank edit screen is presented. If the file exists, the first page is presented on the screen. Text may then be entered or changes made.

At *any* point in the system, the help key (F1 on the PC) may be pressed and a full screen of help is made available. It is not just a list of syntax as with WordStar but a full semantic description of the possible actions at that time. The help processor is fully context sensitive. Pressing the Esc key during the edit presents a choice of editing commands such as copying or deleting, finding or replacing, moving or inserting. The file is saved by pressing Esc and then S. The main menu is then reentered and editing, printing, and spelling checking may then be done. The print format can be set in several ways. The defaults are the screen editing parameters. These can be overridden by print directives in the text, embedded format commands, or console overrides at the time the document is printed.

Installation and Maintenance

SuperWriter comes pre-installed for the IBM PC. A menu-driven program configures the system for a variety of printers – NEC, Diablo, Qume, Xerox, etc. Other than the printer configuration, no other installation is required. A maintenance program is provided with SuperWriter. This program allows Sorcim to distribute fixes for possible bugs so that no programming knowledge is required to apply the fixes. Each fix contains special self-checks so that typographical errors are guarded against. The maintenance program also contains a verification feature to determine if a SuperWriter program has been damaged.

Key Redefinitions

Earlier I promised a method for using some additional PC keyboard keys for common editing functions. This procedure makes use of the key redefinition facility of the ANSI support in PCDOS Version 2.0. You must have the entry DEVICE=ANSI.SYS in the CONFIG.SYS file. I created a batch file named XSW.BAT with the following contents to call SuperWriter:

```
copy swin.key con:
sw
copy swout.key con:
```

The file SWIN.KEY is as follows:

```
<esc>[0;73;18p
<esc>[0;79;21p
<esc>[0;81;3p
<esc>[0;117;2p
<esc>[0;119;20p
```

The file SWOUT.KEY is as follows:

```
<esc>[0;73;0;73p
<esc>[0;79;0;79p
<esc>[0;81;0;81p
<esc>[0;117;0;117p
<esc>[0;119;0;119p
```

The symbol <esc> is the escape character ASCII hex 1B. The file SWIN.KEY redefines the PgUp (Page Up) key to be ^R (Ctrl-R), End to be ^U (End-of-Line), PgDn to be ^C (Page Down), Ctrl-End to be ^B (Bottom-of-File), and Ctrl-Home to be ^T (Top-of-File). The file SWOUT.KEY changes the key redefinitions back at the end of the SuperWriter session.

Conclusion

Sorcim's SuperWriter is a very fine, easy to use word processing package. It deserves a large share of the market. I would recommend the product and its supplier, Sorcim Corporation, very highly.

(Continued on page 120)

BOOK REVIEWS

The Small Computer Connection

By Neil L. Shapiro
Published by Micro Text/McGraw-Hill
$15.95, 190 pages
Reviewed by James Moran*

Subtitled "Telecommunications for the Home and Office," this trade paperback is packed with information on how to access telecommunications services such as CompuServe and The Source with a personal computer.

The author begins with two short chapters on telecommunications basics and quickly moves on to the main purpose of the book – explaining how to use the many services that are available on national databases. This format allows the novice user to become familiar with the concepts of modems and telephone networks, while at the same time allowing the more knowledgeable reader to find needed information without enduring a course on elementary telecommunications.

Obviously a well-informed user of remote computer services, the author uses his knowledge to practically demonstrate the functions of the major networks. Particularly helpful are a number of comparison charts that contrast the services available among networks. This information would be particularly useful to computer users who, although familiar with a particular network, would like to access another on an infrequent basis.

Another nice touch is the abundance of examples that show how to access the various functions within a network. On that basis alone the book would quickly recover its selling price. Had I been able to find a book like this a few years ago, those confusing (and costly) start-up sessions on CompuServe could have been avoided. To paraphrase the author: in telecommunications, time is money.

Throughout the first half of the book, the author speaks primarily of the large commercial networks and their uses. In the second half, the topics become more advanced. There is an excellent section on setting up one-to-one communication between individual computer users. The information presented would be particularly useful where the computers being used are made by different manufacturers. Some helpful suggestions (with examples) on how to handle different computer protocols are made and examples of typical software interfaces are shown. Also included is a short section on implementing a public bulletin board.

The text portion of the book ends with a chapter on advanced techniques for telecommunications. Topics such as database design for bulletin board access are covered here, as are techniques for adding clocking hardware and software for unattended network communications. A particularly intriguing section in this chapter deals with slow-scan television (SSTV) and Telidon videotex. Both of these methods allow for the generation of pictures or graphics on the monitor screen when the computer is telecommunicating.

A useful appendix wraps up the book. Included in it is a well-organized list of public access message systems organized by area code. Although I didn't count them, at least six hundred are listed.

In summary, the book is useful, well written, and understandable. The author, a columnist and editor for *Popular Mechanics* magazine, has put his background to good use. This how-to-do-it guide will be a useful addition to the library of many home computerists.

*This review Copyright © 1983 by Compu-Syn.

Introducing the UNIX System

By Henry McGilton and Rachel Morgan
Published by BYTE Books/McGraw Hill
$18.95, 556 pages
Reviewed by Dr. Joseph B. Rothstein

The jury is still out on the question of Unix for microcomputers, but a verdict should be returned before too much longer. On the one hand, those who are familiar with this operating system/programming environment from its minicomputer implementations are nearly unanimous in praising its power, versatility, and comprehensive set of utility programs. On the other, Unix requires more memory than 8-bit personal computers have generally had available. So it's only with the advent of the new generation of 16-bit chips such as the Motorola MC68000 and Zilog Z8000 that we may see workable implementations of Unix for micros.

Unix is a product of Bell Labs, the research arm of Western Electric and AT&T, and was originally developed for in-house use, particularly for text manipulation and program development. It became such a success among its users that the decision was made to license the system to academic and commercial minicomputer facilities. Judging by the number of microcomputer OEM's and systems houses that have hopped on the Unix bandwagon in just the last few months, it appears that Unix will indeed be a force to be reckoned with in the *micro world* as well.

Though the documentation accompanying the minicomputer versions is voluminous and wide-ranging, personal computer users need and expect more tutorial material than is usually provided by Western Electric and its OEM's. Henry McGilton and Rachel Morgan, the authors of *Introducing the UNIX System*, make no claims that their text is a substitute for existing Unix system documentation; in fact, a disclaimer to that effect appears on the first page of the Preface, under the subchapter heading *Caveat*. But the same Preface describes the intended audience – users who are new to the Unix system – and such an audience is well served in this excellent tutorial.

The first few pages also offer a clear, yet concise overview of the book's organization, along with suggestions on how the reader might best approach reading and using the book. The text divides neatly into two major parts. The first 100 pages cover the essential topics and the key concepts behind the Unix system. This is done in three chapters entitled "Getting Started . . ." "Directories and Files," and "Commands and Standard Files."

The remainder of the book discusses the utilities generally available on Unix systems, and the kinds of things you can do with them. The Unix programmer's "software toolbox" includes electronic mail facilities, text editors and formatters, and program development tools for the C language.

Part of the beauty of Unix is the simplicity of its basic structure; yet its extensibility allows the user to tailor the system to his or her needs and imagination. Even directories – which in most micro operating systems are simple lists of file names and sizes – involve a rich and powerful set of operations in Unix. For example, only one copy of a given file exists on a

multi-user Unix system; other user directories may simply show that file listed, with system pointers to the master copy of the file in question.

Each user has a master directory, which can contain pointers to files created and "owned" by that user or other users, or the directory entry can point to a subdirectory, which may in turn point to files or to another level of subdirectories. Like many Unix concepts, the idea of multi-level directories may be confusing at first, but the authors seem to sense subjects which most need extra explanations, and provide appropriate background material as necessary.

Many of the facilities described are of the sort micro users have only dreamed about. An example is the redirection of input and output, in which the user can specify a file or device as the source or destination of most commands. Something as simple as sending the directory listing to a file for later use is all but impossible on most micros, but it's a snap under Unix.

More significant are facilities to quickly and easily specify a jobstream in which, for instance, the output of one program serves as the input to another, whose output serves as input to a third, whose results are sent to the printer – a process called piping. The user describes the pipes and devices or files, and Unix does the rest. Furthermore, this jobstream can be performed in "background," allowing the user to go on to another task while the system continues to process the jobstream commands until completed, without further intervention by the user.

Unix is not without its problems, and the authors point them out, devoting special attention to those that might most plague the neophyte user. With its large repertoire of powerful commands, it is not difficult to mistakenly tell Unix to do something unexpected or disastrous – and Unix will do little or nothing to warn or stop you.

And since its primary focus is on the multi-user environment, users of personal computers may find many of its facilities cumbersome or of little value. Ideally, some systems house will develop a version of Unix optimized for a single user. New versions of operating systems such as MS-DOS 2.0 (PC-DOS on the IBM) are reported to have "Unix-like" facilities, though they fall far short of full Unix capabilities, which include much more than is described here. Still, it is encouraging to see designers of micro software learning from the best, and despite all her shortcomings, Ma Bell has some bright ideas.

One cautionary note is in order, though it will be obvious to most readers. No text, regardless of its quality, can substitute for hands-on experience and practice. Though I was easily able to follow the discussions and examples in *Introducing the UNIX System*, I didn't for a moment imagine that I would be able to master Unix the first time I tried it. Reinforcement through repeated use will be required before a user feels comfortable with the variety of commands and options available, particularly since so many of the command keywords are compact and have little mnemonic significance (*ls* and *grep* are only two of many examples).

But for an overview of a likely contender in the next round of operating systems wars, *Introducing the UNIX System* seems an excellent place to start. It's an eye-opening tour of a sophisticated approach to human/computer interaction, and it might change your views on the entire field of operating systems.

DDJ

CP/M EXCHANGE

by Robert Blum

I was recently made aware of a series of application notes available from Digital Research that cover problem correction and the addition of some desirable features to CP/M 2.2 and CP/M Plus. I thought they would fit very well into the format of the CP/M Exchange and fortunately so did DRI. As a regular feature I will run one note each month until the supply is exhausted. I think they will provide some useful insight into how CP/M does its job, and may assist you in correcting a problem or two. This month's application note deals with toggling the cntrl-P switch from within a submit job.

Of all the correspondence I receive, questions about interfacing to peripheral devices are the most popular. Most of the questions are about serial channels and adapting them to devices not directly supported by the computer manufacturer. This gray area exists mainly because of the lack of hardware documentation that is provided with today's personal computer. Starting next month I will begin a regular series on how the various manufacturers have implemented their ports to the outside world and any driver software that I can acquire. If you have a problem of this type, drop me a note and I will do my best to answer it.

Intel 8080 to Zilog Z80 Program Conversion

Over the years I have acquired and written many programs that all seem to use a different dialect of the original Intel 8080 opcodes when dealing with the expanded Z80 instruction set. I have been forced to live with this situation because to convert them by hand to Zilog opcodes would take more effort than I was prepared to invest. My dreams of a program to do this job for me was realized when I came upon XLATE2 in the SIG/M library. Ron Conn, author of ZCPR2 and SYSLIB, did an excellent job of producing a program that will automatically convert the standard set of Intel 8080 opcodes to *the Zilog* format with practically no errors. This is definitely the program to have if you want to standardize your assembler source programs as I did.

Hidden Treasures Revisited

Last month I talked about what I thought was the best set of assembler subroutines that I had ever seen. Forget everything I said! After receiving the most recent version of SYSLIB from SIG/M, I was ecstatic to find that it is even better than I thought.

The version that I reported on last month was from an old CPMUG volume. I never did count the number of subroutines contained in that library, but I would guess there were 60 or 70. The documentation was in the form of help files to be used with the on-line help facility also included in that same volume. Well, hold on to your hats wh about the latest version of SYSLIB. .. of subroutines now totals over 1 even counting those that work in conjunction with ZCPR2. The documentation still exists in help file form in addition to two, yes two, manuals that can be printed for hard copy reference. The first manual, SYSLIB User and Reference Manual, is 76 pages long and thoroughly explains what each subroutine's function is and how to use it. The second manual, User's Guide for SYSLIB, is a tutorial on how to effectively use each routine. The documentation as a whole is more than just two pounds of paper and a lot of empty words. It is well written and more than adequately documents what I hold to be the most vital part of all of my program development tools.

One thought hit me soon after reading the table of contents of the first manual. With a good set of macros I could create a high level language of my own design, meeting my own programming goals, while still maintaining the high level of efficiency inherent to assembler programs.

Take it from me, if you are intent on writing assembler programs you need SYSLIB.

CP/M Plus DPB Macro Fix

A few months ago I ran some comments from a reader who had brought up CP/M Plus. One of the problems that he ran into was that the supplied DPB macro would not handle a drive capacity of greater than 8 megabytes. Thanks to William Nichparenko of Aptos, California, there is a short modification that can be made to the DPB macro to alleviate this problem. Bill writes . . .

"Enclosed is a correction for the 8 megabyte limit to the drive size imposed by the Disk Parameter Block (DPB) macro in Digital Research's CP/M Plus system.

"The limit arises when the DSM parameter (related to the file size) is calculated. The simplified expression for DSM is given [in Figure 1 (page 104)].

"This expression has been arranged to minimize multiplications since they are what limit the dynamic range of the DSM parameter. Since RMAC has an upper limit on the arithmetic for expression evaluation of 65536, any product that exceeds this limit creates errors. For example, a 20 megabyte drive would have 2441 tracks and the product of Tracks * SectorsPerTrack = 78125.

"The first assumption that is made is that the block size is an integral multiple of the sector size. This is true for all current drives that I know of, and CP/M would not be able to function if this were not true. This implies that the expression (BlockSize/SectorSize) will always be a whole integer.

"One interesting aside is that just the simple rearrangement of the expression for DSM highlights the fact that the upper limit on the drive size is sector size dependent. Thus, the limit is 8 megabytes for a sector size of 128 bytes, 16 megabytes for 256 byte sectors, etc.

"What is really needed is some simple method for RMAC to do 32-bit arithmetic. I am indebted to Dan Ross of Succinct Systems for suggestion of the following algorithm.

"Before the multiplication of system tracks times the number of sectors per track is performed, each variable is split into a high and low byte, and the product then done. The resulting expression is only slightly more complicated than the original, but will allow the macro to function up to the 512 megabyte drive limit inherent in CP/M Plus.

"Let

```
(TotalTracks - Offset) = (Tm*256 + T1)
SectorsPerTrack        = (Zm*256 +Z1)
```

where Zm is the most significant byte of the 16-bit SectorsPerTrack and Z1 is the least significant byte of the 16-bit SectorsPerTrack with Tm and T1 defined in a similar manner for the track variable. The *product of the tracks* times sectors now becomes:

```
(Tm*256+T1)(Zm*256+Z1) =
    Tm*Zm*256*256 +
        (TmZ1 + T1Zm)*256 + T1Z1
```

"This expression is still very unwieldy since we must be content with the product 256*256 wich is already 65536.

Thus, we need to make one more simplifying assumption, that the number of sectors per track is never greater than 256, thus Zm = 0. This results in the very simple expression for DSM:

DSM = Zm*Z1*(256/Mblk) +
(T1*Z1)/Mblk - 1

where Mblk = BlockSize/SectorSize.

"This is a very reasonable assumption to make since all drives that I currently know of have less than 256 sectors per track. Since this is a possible source of error in the far future, the macro can provide a warning to the user if Zm is not zero. In addition the macro will provide a warning to the user if DSM exceeds the maximum value that CP/M Plus can accommodate, which is 255 for 1K block size and 32767 for 2K to 16K block sizes."

Listing One (page 105) contains his changes to the current DPH macro that need to be made to implement the above correction.

Quicker Submit File Truncation

David Kirkland of Houston, Texas, recently wrote to share a quicker way to truncate records from the end of a submit file. Dave writes . . .

"I was interested to read the item concerning the SKIPIF command in the August CP/M Exchange because I had recently been working on a similar program myself.

"There is a quicker (but somewhat sneakier) way to truncate the last record from the active $$$.SUB file in order to skip the next command. John Ramsey copied all of the file except for the last record to produce the truncated file. The quicker way is to coax the BDOS to truncate the file in place by having the directory entry for the file altered. One problem with this method is that it does not work for files of more than one extent (128 records). However, the CCP (and ZCPR) will not properly handle a $$$.SUB file consisting of more than 128 records anyway, so it's no great loss. I don't know if the trick of modifying the system byte in the FCB to cause the BDOS to write the FCB back to the directory on a close is common knowledge, but that's what the CCP and ZCPR do themselves.

"I have found that the SKIPNIF and QUITIF functions combined with nested submits allow me to do all sorts of useful things easily; quite a step from the original bare-bones submit functions. I also would recommend one further change that I've made to my system. I put the $$$.SUB file on a RAM disk. This one step can save a lot of waiting and a lot of wear and tear on your disk drive arm, to boot."

Dave's comment about zeroing the S2 byte of the FCB to mark it for writing to the directory is characteristic of a few internal tricks that can be used with CP/M. One reason that they are not commonly known is that they are not documented. Which brings up a subject that must be considered before charging ahead and using trick code. DR purposely omitted documentation on some of the features of CP/M because they are considered internal techniques only to be used by the operating system. And since they aren't documented, that means they may not work in subsequent releases. The fact of the matter is this: if you use it and later it suddenly stops working, don't gripe. DRI is only responsible for what is documented, not what the hacker has found out by prodding into the object code.

I must admit that I am as guilty as the next person of finding and using tricks, but I have confidence that, if something suddenly changes, finding a new way can't be any more difficult than finding the original trick. Those are famous last words if I ever heard them. The choice is yours.

Dave's changes for John Ramsey's SKIPIF program are contained in Listing Two (page 106). Next month I will run Dave's SKIPNIF program.

CP/M Application Note 14

For: CP/M 2.2

Copyright © Digital Research, Inc. 1982. All rights reserved. Text adapted from the original.

When using the SUBMIT facility for batching job streams, no facility is provided to turn the CTRL-P toggle on and off from within the SUBMIT file. Even though there is a patch for SUBMIT that forces it to accept control characters when preceded by a circumflex, ^, SUBMIT does not act on a CTRL-C or a CTRL-P.

The assembly language program that follows, CNTLP (Listing Three, page 106), will turn the CTRL-P function on or off within a SUBMIT file. The same assembly code can also be used from within an application program or when loaded and saved/executed directly from the console to perform the same function. If XSUB is to be active in the SUBMIT file, CNTLP must be executed before XSUB. Even though the CP/M V2.2 Operating System Manual states, "The XSUB command is included as the first line of your submit file. . .," this is not necessary. If it is necessary to use XSUB and also to alter the CTRL-P status multiple times in the same submit procedure, XSUB can be deactivated by using the assembly language program in Listing Four (page 109). This program also turns off CTRL-P if it is on when used.

When using CNTLP, if the ^P toggle is off, it is turned on; if on, it is turned off. An appropriate message is output to the console and also echoed to the list device indicating the CTRL-P function status.

CNTLP calculates the location of the ^P toggle byte in the BDOS by first getting the BDOS address from the jump instruction located at 05H and adding 02FEH to that address. At this address there must be a verifiable four-byte sequnce or an error message is output to the console indicating that the BDOS could not be found. As an operational note here, if this occurs it means that some other program exists, such as DDT or XSUB, that has already storage-protected itself by altering the BDOS entry point at location 05H. The calculated address is then altered to point to the CTRL-P function byte, and an exclusive or is performed on it to turn ^P on or off depending on its initial status.

DDJ

(Listing begins on page 105)

DSM = ((TotalTracks - Offset) * SectorsPerTrack)/(BlockSize/SectorSize) - 1

Where	TotalTracks	=	Total number of usable tracks on the drive
	Offset	=	Number of tracks allocated for system use
	SectorsPerTrack	=	Total number of sectors per track
	SectorSize	=	Total number of bytes per sector
	BlockSize	=	CP/M defined block size (1024, 2048, etc.)

Figure 1.

CP/M Exchange (Text begins on page 102)

Listing One

```
;First add a declaration of new local variables and maybe a few lines
;     telling the reasons for the change and a reference to documentation
;     explaining the change at greater length.

          LOCAL     Tm,Tl,Zm,Zl,Mblk

;Next define the variables

Tm        SET       HIGH (?trks-?off)
Tl        SET       LOW  (?trks-?off)
Zm        SET       HIGH ?pspt
Zl        SET       LOW  ?pspt
Mblk      SET       ?bls/?psize

;Add warning for too many sectors per track

          IF        Zm NE 0
          'Warning, DSM parameter invalid due to sectors/track > 256'
          ENDIF

;Replace previous calculation for ?dsm with the updated calculation
?dsm      set       Tm*Zl*(256/Mblk) + (Tl*Zl)/Mblk - 1

;Add warning for drive size too big

          IF        ?bls NE 1024
                    IF        ?dsm > 32767
                    'Warning, Drive size too big for CP/M'
                    ENDIF
          ENDIF
```

End Listing One

Listing Two

```
OPEN      EQU       15              ;BDOS OPEN FUNCTION
CLOSE     EQU       16              ;BDOS CLOSE FUNCTION

SKIP      EQU       $

          MVI       C,OPEN          ;OPEN $$$.SUB FILE
          LXI       D,SUBFILE       ;
          CALL      BDOS            ;

          INR       A               ;CHECK IF $$$.SUB EXISTS
          RZ                        ;RETURN IF NOT

          LXI       H,SUBFILE+14
          MVI       M,0             ;ZERO THE S2 BYTE OF THE FCB TO FORCE
                                    ;A WRITE OF THE MODIFIED FCB BACK TO
```

```
                                ;THE DIRECTORY. NOW POINT TO THE RC
                                ;FIELD OF THE FCB TO DECREMENT THE
                                ;NUMBER OF RECORDS IN $$$.SUB.
        INX     H
        DCR     M

        MVI     C,CLOSE         ;CLOSE $$$.SUB TO REWRITE THE MODIFIED
        LXI     D,SUBFILE       ;FCB
        CALL    BDOS
```

End Listing Two

Listing Three

```
;
;       Program to turn list device on and off from SUBMIT procedure
;

BDOS            EQU     5H              ;BDOS JUMP INSTRUCTION
BDOSA           EQU     BDOS+1          ;BDOS ENTRY ADDRESS
PSTR            EQU     9               ;PRINT STRING FUNCTION
TESTOFF         EQU     02FEH           ;OFFSET FOR VERIFICATION
LISTCP          EQU     0DH             ;^P OFFSET IN PAGE
MVIC            EQU     03EH            ;MVI C,X INSTRUCTION
CR              EQU     0DH             ;CARRIAGE RETURN
LF              EQU     0AH             ;LINE FEED

CNTLP:
                ORG     0100H

                LHLD    BDOSA           ;PICK UP ADDRESS OF BDOS IN HL
                LXI     D,TESTOFF       ;OFFSET OF ^P PAGE IN DE FOR ADD
                DAD     D               ;HL= COMPARE AREA IN BDOS
                LXI     D,STRING        ;DE= COMPARE STRING

COMPARE:
                LDAX    D               ;IS CHARACTER A 0?
                ORA     A               ;
                JZ      OK              ;YES , WE'RE DONE
                CMP     M               ;IS BDOS SAME AS STRING
                INX     H               ;NEXT BYTE
                INX     D               ;
                JZ      COMPARE         ;BDOS = STRING KEEP LOOPING

ERROR:
                MVI     C,PSTR          ;ELSE PRINT ERROR MESSAGE
                LXI     D,ERRORMSG      ;
                JMP     BDOS            ;RETURN TO CCP FROM BDOS

OK:
                MVI     L,LISTCP        ;LISTCP PAGE OFFSET
```

(Continued on page 108)

 (Listing continued, text begins on page 102)

Listing Three

```
                MVI     A,1             ;TOGGLE ^P BYTE ON OR OFF
                SUB     M               ;TRUE = 1, FALSE = 0
                MOV     M,A             ;PUT RESULTS BACK IN MEMORY
                CPI     00H             ;SEE IF ON OR OFF
                JZ      OTHERMSG        ;TO ISSUE APPROPRIATE MESSAGE
                LXI     D,ONMSG         ;^P TURNED ON
                JMP     PRINT           ;GO AROUND

OTHERMSG:
                LXI     D,OFFMSG        ;^P TURNED OFF

PRINT:
                MVI     C,PSTR          ;PRINT SIGN-ON MESSAGE
                JMP     BDOS            ;RETURN TO CCP FROM BDOS

ONMSG:          DB      CR,LF,'(^P turned on)$'
OFFMSG:         DB      CR,LF,'(^P turned off)$'
ERRORMSG:       DB      CR,LF,'Unable to find BDOS$'


STRING:         DB      RET,MVIC,1,JMP,0

                END
```

End Listing Three

Listing Four

```
;
;       Program to Deactivate XSUB and Reset ^P function
;

DEXSUB:

                ORG     0100H

BDOS            EQU     05H             ;BDOS LOCATION IN BASE PAGE
CR              EQU     0DH             ;CARRIAGE RETURN
LF              EQU     0AH             ;LINE FEED

                MVI     C,9             ;PRINT STRING = FUNCTION 9
                LXI     D,MSG           ;
                CALL    BDOS            ;

                MVI     C,0             ;SYSTEM RESET = FUNCTION 0
                JMP     BDOS            ;EXIT BDOS TO CCP

MSG:            DB      CR,LF,'(XSUB deactivated; ^P turned off if on)$'

                END
```

End Listing Four

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

IBM PC Technical Reference

IBM has quitely released a drastic revision of the popular Technical Reference Manual for the Personal Computer. It can be recognized on the store shelves because of a white "2.02" sticker on its spine, and weighs in at some 660 pages as opposed to the older version's 360 odd pages. Most of the sections have been expanded; material on the Intel Numeric Coprocessor has been added; the BIOS listings for the hard disk driver are included; and the tables and schematics for the various new expansion cards and the new 256 Kbyte mother board are also present. There is no provision for buying an updated page set for your old manual . . . you'll just have to spend the $36 to get another one.

We don't have room in this month's column to describe all of the interesting new additions to the Technical Manual, but here is one example. On page 2-11, tucked away in the back next to the ROM BIOS listing, is a description of a hitherto undocumented feature of the system that is used to install the hard disk driver at cold start time. It is titled "Adapter Cards with System-Accessible ROM Modules," and describes how after the BIOS sets up its interrupt vectors during the initialization sequence, it scans the memory space from 0C8000H to 0F4000H in 2 Kbyte blocks looking for valid ROM software.

A valid ROM routine is defined to begin as follows:

byte	contents
0	055H
1	0AAH
2	length indicator

The length indicator represents the number of 512-byte blocks in the ROM, i.e., length/512. A checksum over the code in the ROM is also done by the BIOS scan routine; each byte is summed modulo hex 100 and the total must be zero. Assuming the ROM passes these validity tests, the BIOS performs a far call to byte 3 of the ROM, which must therefore be executable code. The ROM is then free to take over system interrupts as needed in order to substitute its own drivers for the normal system device drivers. Finally, the ROM should return control to the BIOS initialization sequence via a "Far Return."

This is really a nifty capability. It can be used by third party expansion card manufacturers to provide enhanced video graphics cards or disk controllers with the driver software in PROM, rather than having to supply driver files on diskette and count on the user to install them properly. It can also be used by *you*, the sophisticated PC owner and loyal reader of this column, to permanently install your own software inventions, test routines, character tables, and who knows what else into those empty EPROM slots on the mother board.

The IBM PC Exhibitions

I was tempted to include a few comments here after the PC-83 Show which was put on in June by Northeast Exhibitions in San Francisco, but decided to desist so as not to prejudge the PC Faire by Jim Warren's organization that was scheduled for August. But now it can be told, since both shows are safely over with – they were both duds.

The number of attendees was low for both, and with good reason: there was very little that was interesting to see. All the PC oriented magazines had their traditional booths, as did the disk media manufacturers, the RAM expansion board builders, the DBASE II utility sellers, the discount software mail order marketers, etc. – yawn . . . .

But booths displaying innovative hardware products or novel new programs were scarce as could be. In fact, the only booth I discovered at PC Faire which made the trip worthwhile belonged to Morgan Computing Co. They were demonstrating their new "Professional BASIC," which allows full use of the 8088 memory space and includes 8087 support, syntax checking at entry time, a window-oriented debugger, conditional breakpoints, and upward compatibility with IBM PC Basic. This product makes Microsoft Basic look pretty old-fashioned. Interestingly, Professional BASIC was offered to both Microsoft and Digital Research by its author, and was turned down – perhaps both companies think they are making enough money already! Morgan was also showing a very impressive window-oriented assembly language debugger called "Trace-86," which we will cover in greater detail in an upcoming column.

68000 News

We have finally received copies of two different CP/M-68K implementations for the CompuPro 68000 system, one by Gifford Engineering and the other by CompuPro itself. The contrast between the two is quite striking. Gifford chose to write all of the necessary utilities and the CP/M BIOS in C; CompuPro did the BIOS using the traditional assembly language approach. Perhaps it's a coincidence, but the Gifford implementation works very smoothly while the CompuPro release is ridden with severe bugs: the system fails to boot from distribution disk #1 as described in the manual, printer support for the Interfacer 4 card does not work, and the one of the console status BDOS functions for the Interfacer 4 card hangs up the system irretrievably – to name but a few.

On another subject altogether, we recently received an outraged letter from a Cromemco 68000 system purchaser who had divined why the performance of the 68000 Cromix operating system is so poor. It turns out that most of Cromix is still compiled to Z80 machine code, and the 68000 part of the dual CPU card is only in use while the application program is executing. With this little tidbit in mind, you can go back and read the Cromemco advertisements about their super-duper enhanced Cromix for the 68000 with a new outlook.

Floating Point Benchmarks

The floating point benchmark figures from Bill Savage that I included in the September column have generated a great deal of mail from readers. In a few months I will gather together all of the additional contributed benchmark results and publish them in a new table. In the meantime, I am providing a Pascal and Forth version of the same benchmark so that everyone's figures will be comparable (see Listing One, page 112).

MS-DOS COM File Loader

This month, I'm including a useful utility called BIGLOAD written by Rick Wilton for MS-DOS users (Listing Two, page 112). It turns out that MS-DOS uses the "block read" function to load a COM file, and can't handle such a file if it's larger than 65535 bytes (although an EXE file of almost any size can be loaded successfully). Some languages, such as our PC/FORTH+ package, allow the user to write a new executable image with their

own precompiled extensions, and files of this type can get quite large. BIGLOAD allows you to load such a COM file, optionally supplying a second filename *which will* be filled into the default file-control-block at 005CH.

BIGLOAD illustrates a number of useful techniques for MS-DOS assembly language programmers: using the block read functin 27H, setting up a new program segment prefix and terminate address, and transferring control to a "spawned" program. The latter two functions can be done more simply under DOS 2.0 via the new "Load or Execute a Program" service, function 04BH, but this method was not used in BIGLOAD in order to preserve compatiblity with DOS 1.0 and 1.1.

DDJ

(Listings begin below)

16-Bit Toolbox (Text begins on page 110)

Listing One

```
PASCAL  version,  contributed  by  Jeffrey  Speiser  of  San  Diego,
California:


PROGRAM SAVAGE(INPUT,OUTPUT);
{SPEED AND ACCURACY TEST FOR FLOATING POINT}
{REF: DR.DOBB'S JOURNAL, SEPT. 1983, PP. 120-122}
CONST
        ILOOP=2500;
VAR
        I:INTEGER;
        A:REAL8;
{TNDRQQ IS DOUBLE PRECISION TANGENT}
FUNCTION TNDRQQ (CONSTS A: REAL8):REAL8 ; EXTERN;
BEGIN
  WRITELN('START');
      A:=1.0;
        FOR I:=1 TO ILOOP-1 DO
           BEGIN
               A:=TNDRQQ(ARCTAN(EXP(LN(SQRT(A*A)))))+1;
           END;
  WRITELN('A=',A :20:14);
  WRITELN('DONE');
END.


FORTH  version,  as  run on PC/FORTH with 8087 floating point support.
This  uses the 8087 functions in the simplest mode with no chaining of
operations on the 8087's internal stack:


: (FN)      FDUP F* FSQRT FLN FE**X FATAN FTAN ;
: FN        2500 1 DO (FN) 1. E 0 F+ LOOP ;
: SAVAGE    1. E 0  FN  CR   ." Result: " F. ;
```

End Listing One

Listing Two

```
name    bigload
page    55,132
title   'BIGLOAD -- load .COM file bigger than 64K'
```

```
                        ;
                        ;
                        ; BIGLOAD -- load .COM file bigger than 64K
                        ; ( for IBM PC under PC-DOS 1.1 or 2.0 )
                        ; Version 1     March 1983
                        ;
                        ; Richard Wilton
                        ; Laboratory Microsystems
                        ; 4147 Beethoven Street
                        ; Los Angeles, CA  90066
                        ;
                        ;

0000                    cseg    segment byte
                                assume  cs:cseg,ds:cseg
0000                    bigload proc    far             ; sets up far return ...
0100                            org     100h

= 000D                          cr      equ     0dh             ; carriage return
= 000A                          lf      equ     0ah             ; line feed

0100  BA 01AF R         start:  mov     dx,offset mes0  ; startup message
0103  B4 09                     mov     ah,9
0105  CD 21                     int     21h
0107  33 D2                     xor     dx,dx           ; zero DX
0109  B4 25                     mov     ah,25h          ; set terminate address ...
010B  B0 22                     mov     al,22h          ; ... for new program segment
010D  CD 21                     int     21h

010F  BA 0203 R                 mov     dx,offset endofs ; offset to end of this loader
0112  B1 04                     mov     cl,4            ; no of bits to shift
0114  D3 EA                     shr     dx,cl           ; convert byte addr to paragraph
0116  42                        inc     dx              ; offset of 1st available segment
0117  8C C8                     mov     ax,cs           ; current segment to AX
0119  03 D0                     add     dx,ax           ; actual value of 1st available segment
011B  89 16 0201 R              mov     useg,dx         ; save it for later ...
011F  8E C2                     mov     es,dx           ; ... and for subsequent move
0121  B4 26                     mov     ah,26h          ; call to DOS
0123  CD 21                     int     21h             ; create new program segment

0125  BE 006C                   mov     si,6ch          ; 2nd param FCB in current segment
0128  BF 005C                   mov     di,5ch          ; 1st param FCB in new segment
012B  B9 000C                   mov     cx,0ch          ; byte count for move
012E  F3/ A4                    repz movsb              ; copy the filename

0130  8C C8                     mov     ax,cs           ; copy current code seg ...
0132  8E D8                     mov     ds,ax           ; ... to DS
0134  BA 005C                   mov     dx,5ch          ; DS:DX points to FCB of .COM file
0137  8B DA                     mov     bx,dx           ; make FCB addressible
0139  C6 47 09 43               mov     byte ptr  9 [bx],43h ; force a "C" ...
013D  C6 47 0A 4F               mov     byte ptr 10 [bx],4fh ; ... an "O" ...
0141  C6 47 0B 4D               mov     byte ptr 11 [bx],4dh ; ... and an "M" into the filename
0145  B4 0F                     mov     ah,0fh          ; open the .COM file
0147  CD 21                     int     21h
0149  0A C0                     or      al,al           ; test return code
014B  75 41                     jnz     exit1           ; exit if non-zero
014D  C7 47 21 0000             mov     word ptr 33 [bx],0000 ; zero the random ...
0152  C7 47 23 0000             mov     word ptr 35 [bx],0000 ; ... record field in the FCB
```

16-Bit Toolbox (Listing continued, text begins on page 110)

Listing Two

```
0157  A1 0201 R                    mov     ax,useg         ; copy the new segment into ...
015A  8E D0                        mov     ss,ax           ; ... reg SS
015C  BC 0100                      mov     sp,100h         ; now SS:SP can't be trampled on
015F  8B 1E 0201 R                 mov     bx,useg         ; initialize BX to keep track ...
0163  83 C3 10                     add     bx,10h                  ; ... of the current DTA
0166  8E DB                        mov     ds,bx           ; set up DS:DX to point to the DTA
0168  33 D2                        xor     dx,dx
016A  B4 1A                        mov     ah,1ah          ; set up DOS call and do it
016C  CD 21                        int     21h

016E  B9 0100              loop1:  mov     cx,100h         ; number of records of length 80h
0171  8C C8                        mov     ax,cs           ; copy current CS to DS
0173  8E D8                        mov     ds,ax
0175  BA 005C                      mov     dx,5ch          ; DS:DH points to FCB of .COM file
0178  B4 27                        mov     ah,27h          ; do random block read
017A  CD 21                        int     21h
017C  A8 01                        test    al,1            ; end of file?
017E  75 17                        jnz     exit2           ; yes, so exit

0180  81 C3 0800                   add     bx,800h         ; increment location of DTA
0184  8E DB                        mov     ds,bx           ; copy to DS
0186  33 D2                        xor     dx,dx           ; DS:DX now points to next DTA
0188  B4 1A                        mov     ah,1ah          ; set up DOS call to set DTA
018A  CD 21                        int     21h
018C  EB E0                        jmp     loop1           ; do it again

018E  BA 01D2 R            exit1:  mov     dx,offset mes1  ; "file not found"
0191  B4 09                        mov     ah,9            ; write to terminal
0193  CD 21                        int     21h
0195  CD 20                        int     20h             ; exit to DOS

0197  8C C8                exit2:  mov     ax,cs           ; copy CS to DS
0199  8E D8                        mov     ds,ax
019B  BA 01EA R                    mov     dx,offset mes2  ; "file loaded"
019E  B4 09                        mov     ah,9            ; write message to terminal
01A0  CD 21                        int     21h
01A2  A1 0201 R                    mov     ax,useg         ; set up registers for new segment

01A5  8E D8                        mov     ds,ax
01A7  8E C0                        mov     es,ax
01A9  50                           push    ax              ; push new CS onto stack
01AA  B8 0100                      mov     ax,100h
01AD  50                           push    ax              ; push offset onto stack
01AE  CB                           ret                     ; FAR return causes CS:IP to be set

01AF  0D 0A 44 4F 53 20    mes0    db      cr,lf,'DOS 2.0 Multi-segment loader ...$'
      32 2E 30 20 4D 75
      6C 74 69 2D 73 65
      67 6D 65 6E 74 20
      6C 6F 61 64 65 72
      20 2E 2E 2E 24
01D2  0D 0A 0D 0A 2E 43    mes1    db      cr,lf,cr,lf,'.COM file not found$'
      4F 4D 20 66 69 6C
      65 20 6E 6F 74 20
```

```
      66 6F 75 6E 64 24
01EA  0D 0A 0D 0A 2E 43       mes2    db      cr,lf,cr,lf,'.COM file loaded',cr,lf,'$'
      4F 4D 20 66 69 6C
      65 20 6C 6F 61 64
      65 64 0D 0A 24
0201  0000                    useg    dw      0

= 0203                                endofs  equ     $
0203                          cseg    ends                    ; end of code segment

                                      end     start           ; end of assembly -- start addr at "start"
Open procedures:

BIGLOAD. . . . . .
```

End Listing Two

OF INTEREST

by Michael Wiesenberg

Product Codes for Coded Products

Software will be vended like groceries if **Softwareland** has its way. Softwareland urges "software vendors to adopt uniform product codes, such as the grocery industry has already done." Software products will carry Uniform Product Codes (UPCs) – those familiar bar codes we see on our milk cartons, candy bars, etc. These have the advantage of keeping the retailer posted on stock movement and reduce the kinds of errors caused by a cashier's incorrect number entry. Electronic point-of-sale (POS) systems keep track of inventory. Softwareland is aggressively pursuing this policy in mass mailings to software vendors, whom they assist by producing free camera-ready copies of the UPC bars and numbers. For their own POS systems, Softwareland uses magnetic strip readers for credit card sales which automatically check a card's validity by telephone communication with the issuing bank. Apart from the appropriateness of such a scheme's onset (now that we're getting so close to 1984), I wonder if they've thought it all through. Read a magnetic disk product with an electronic bar scanner? Won't that affect the data? Softwareland currently has stores in Scottsdale, Phoenix, and Mesa, Arizona, and plans to open one per month, starting immediately, with several hundred open by 1986. **Reader Service No. 101.**

The Robots Are Coming

Robots may soon be as commonplace in homes as microcomputers. **Androbot** announces several personal robots that entertain, communicate, and do other useful things. **B.O.B.** (Brains on Board) has an 8086 board and 3Mb of memory. Infrared sensors attract it to people, whom it follows, using ultrasonics to avoid objects in its way, while talking in a vocabulary of over 100 words and phrases. B.O.B. can carry things for you, or pull them around in its optional AndroWagon. It can fetch a beer from an optional AndroFridge. Through its Androbus system, it can be expanded, as well as through commercially available software (which is created by the user with the Androbot Control Language – ACL). To program B.O.B. it must first be attached to any computer or terminal with a serial port. Ten slots house electronic components for the operating system and permit additional cartridges and plug-in boards for memory and CPU expansion, voice recognition, text-to-speech modules, etc. While I haven't put a "TM" next to all of those "Andro" terms, Androbot has carefully trademarked all of them, including other options like the AndroSentry home security and alarm package. B.O.B. is 36.5 inches tall and 24 inches wide, weighing 42 pounds. It runs on rechargeable batteries that permit several hours between charges, and costs $2995. AndroWagon is $95.

B.O.B. has a little sibling called **Topo**, with fewer capabilities and a lower price tag – only $795.

The baby of the family is **F.R.E.D.** (Friendly Robot Educational Device), a pint-sized version which can be programmatically controlled, or with a remote infrared controller. Attach a pen to F.R.E.D. and it draws on paper. F.R.E.D. comes with a mini AndroWagon so that it can move small items about. This one costs under $300.

AndroMan is billed as the "world's first real-life three-dimensional videogame robot. Use it with Atari VCS 2600 or compatible machines. The foot-high miniaturized Androbot is controlled by joystick with a remote infrared signal and comes with game cartridge, transmitter, game playing field, and game pieces imprinted with coded information. Navigate AndroMan through an obstacle course after beginning play on screen. Contact with coded game pieces triggers further interactions. During game play, AndroMan talks to players. We weren't told a price on this one, but it, like all the others, was "designed with 'expandability in mind.'" **Reader Service No. 103.**

PC Ports, Clocks, and RAM

RAM+3 from **Seattle Computer** is a slot-saving, multifunctional card for IBM PC and XT with DOS 1.1 or 2.0. It has clock/calendar, battery backup, parallel and RS-232 serial ports, up to 256K additional RAM, and Flash Disk drive simulation software. Seattle Software claims that because they are the original developers of PC DOS, they are experts at properly implementing the software, which reduces runtime of disk-intensive programs by as much as 80%. A nonexpandable board is $210, while a version with sockets but no additional memory is $320. Adding 64K brings the price to $395; 128K to $470; 192K is $545; 256K is $620; and expansion 64K chip kits are $80. **Reader Service No. 105.**

Dual-Port Daisywheel

The **Image I** daisywheel printer from **Primages** has switch-selectable RS-232 and Centronics connections and an automatic sheet feeder. It prints at 45 cps and sells for under $2000. Looks good, but what they don't tell us is whether it offers more than one pitch, subscripts, superscripts, proportional printing, etc. I suspect these are included, but you'd better ask. **Reader Service No. 107.**

A Carryover

Send files between any two CP/M machines that can be physically connected (*including* Apple CP/M) (either by RS-232 ports or by two modems) with **The Transporter**, from **Workman and Associates**. According to Workman, you need this product because there is no way an 8-inch drive can read a 5¼-inch drive, and there are no standard 5¼-inch formats. Even two machines with the same physical disk requirements can't talk to each other, so KayPros and Osbornes can't read each other's disks. The Transporter does not have to be installed, nor do you need different copies for each computer. It sends itself and then can send from the second machine. It sends to CP/M-86. A CP/M-86 sender will soon be available. $69.50.

Workman and Associates also offers what many call the best version of C for the price, **BDS C**, at lower than list – $130. This version of C compiles faster than most (notably the best-known version that sells for over $500), has good error recovery, and generates tight code. While it is not "full" version 7 compatible C, lacking enumeration types and explicit reals, "most everything works á la Kernighan and Ritchie." A reals package is included as an addendum. Add $10 for the standard Kernighan and Ritchie text and $15 for a disk of example programs, including games, CP/M low-level utilities, a file listing program, and the SQ/USQ file compacting programs. Updates cost $35 for those who have earlier versions than 1.50 of BDS C. Return the original disk and get it and two new disks back (one with debugger). **Reader Service No. 109.**

Your Pal SAL

SAL/80 and **SAL/86** by **Protools** are Structured Assembly Language compilers that compare to C or Pascal in emitting optimally dense test/branch code for a vocabulary of control structures. I/O primitives permit complex, user-friendly interfaces to a level comparable to BASIC. A compiler toggle enforces optimal use of Z80 instructions. While SAL currently supports Intel mnemonics, Z80 is coming soon in an update. Protools describe this as the ideal tool for engineers and programmers who want to cut development costs in half and reduce maintenance costs by 90%. You need CP/M, MAC/RMAC, 64K, at least 140K disk space, and $100. **Reader Service No. 111.**

What's a VIXEL?

VIXELs are a series of $12.95 tape packages from **The Code Works** for the VIC-20, most of which work on the unexpanded machine. Among the games are: **Fire**, a helicopter-putting-out-fires game; **Draw**, a high-res sketchpad program; and **Race**, a "smash-em-up" game of chicken. Another game is called **Warp**, which simulates the feeling of flying in a spaceship through a hyperspace tunnel (requires joystick).

The Code Works also offers **Cursor 64**, $12.95 tape packages for the Commodore 64. Among the games in this package are: **Fifteen**, **Safe**, wherein you must open a safe with rotating combination lock before the cops come to get you; and **Piano**, with a keyboard and a bouncing dot (a sprite) which shows the keys being pressed, and comes complete with three tunes and instructions for adding more of your own. Add $1 per order for postage and handling; Californians add 6% sales tax. **Reader Service No. 113.**

Indecipherable Software

Dedicate/32, from **Merritt Software**, is RSA (Rivest Shamir Adleman) based public key encryption software for Z80 CP/M or MP/M microcomputers. It can be used by businessmen to protect payroll and client records, electronic mail subscribers, security managers, programming departments to avoid disclosure of development paths, bankers wishing to protect EFT, and companies transmitting confidential information to branch offices. The public key system is RSA type, modulus size 256, with an encryption time of 33 seconds, and totally random and unpredictable key generation produced in under 15 minutes. This company claims that Dedicate/32 destroys the "myth" that there is no such thing as a truly random number (theirs is generated by disk latency and Z80 refresh counter). The single key system is type PKS Chaos Engine A-81, key size 255 bytes, with a speed of 9600 baud, and a block size of 256 bytes. The integrated system encodes any CP/M file with the output able to be sent by any communications system. The public key technique means never having to reveal your decrypting key, for extra security. Merritt claims Dedicate/32 is uncrackable. I wonder if they warranty any financial losses occasioned by data loss if the code *is* ever cracked? $175 plus $5 (surface shipping) or $6 (air shipping). **Reader Service No. 115.**

A Program for 16 Computers

Hobbyscoop (pronounced "Hobbyscope") started as a Dutch concept. Originally for four computers, each week software was broadcast for a different computer in a form that could be input to the tape interface. The interfaces were too slow for two of the machines, necessitating eight minutes at a time of horrible on-the-air noises. The interfaces were not reliable; and, consequently, the interests of more computer enthusiasts were aroused. This led to the development of NOS-BASICODE, a language which can be read by over 16 computers. You order a handbook explaining NOS-BASICODE, and a cassette (readable by any of the current group of computers) that "teaches" BASICODE to your computer through a translation program. Through an international radio network, the Dutch-and-English broadcast of this material can be heard throughout much of the world. Computers which are capable of reading NOS-BASICODE are Apple, BBC microcomputers, all Commodore computers (including PET and VIC), CP/M computers, DAI, Exidy Sorcerer, Heath Zenith, OSI Challenger, Philips P2000, SWTPC-6800, and TRS-80 Model I and III. You can pay in two ways: either transfer the amount by the Postgiro system to account number 1419 in Hilversum, or send an international money order (IMO) for the exact amount payable to "Nos Algemeen Secretariaat." In both cases, payment *must* be in Dutch guilders. Prices, all in Dutch guilders, are: The Netherlands, 25; Europe, 30; outside Europe seamail, 30; U.S. and Canada airmail, 35; Republic of South Africa airmail, 38; Japan airmail, 38; Australia and New Zealand airmail, 38. Payment goes to BASICODE, Administratie Algemeen Secretariaat, NOS, P.O. Box 10, 1200 JB Hilversum, The Netherlands. Send any correspondence (English or Dutch) regarding BASICODE in general, or program contributions, to Hobbyscoop, P.O. Box 1200, Hilversum, The Netherlands. Send any inquiries about Radio Netherland's English service or regarding the "Radio-Activity" program to Jonathon Marks, "Radio-Activity," Radio Netherlands, Transcription Service, P.O. Box 222, 1200 JG Hilversum, The Netherlands.

DDJ

VERSA-FILE

Company: Analytic Management Systems, 247 High Street, Palo Alto, CA 94301

Computer: CP/M 2.2 or greater, and MP/M

Price: $69.00

Circle Reader Service No. 133

Reviewed by Daniel R. Lunsford

VERSA-FILE is a small data file management package with full-screen data entry and display, built-in report generation, sort package, and other maintenance utilities. It is limited in scope; for instance, files are limited to four character keys and only 1000 records per file. Even with these limitations, however, I found it did what I wanted with reasonable speed and no discernible bugs (once I got it up and running, that is – getting it running was another thing altogether!). My overall impression is that of a good solid program that does what it was designed to do. It won't do everything, but what do you expect for $69?

Operation of the package is simplicity itself. The structure of each file is described in a "schema" file, which is prepared with a standard text editor. The schema contains the name of each field in a record, its attributes (length, datatype, where on the screen the field is to be displayed, etc.), and the prompts that will be displayed onscreen during an entry session. Once this is prepared, a VERSA-FILE utility is run, which creates the indexes, and screen formats, and displays the actual datafile. Then you just enter data. It's completely menu driven, and catches most data-entry errors as well.

The system includes select and sort logic, which I found very easy and natural to use. As an experiment, I created a book-cataloging schema for our library, stuck about 100 books into the database, and started selecting and sorting for various criteria; it performed flawlessly each time. The limitations are within reason for a package of this size and price range; the sort, for instance, is an in-memory variety and hence cannot handle a really big file. I estimate that my library would need at least five VERSA-FILE data sets (I have a *large* library). Again, not bad for the price.

The report generator facility is one of the simplest and best I've ever seen. I once had the occasion to see someone use another data manager, one of the large $500+ ones, and getting reports out of it was like pulling teeth. VERSA-FILE, on the other hand, has one of the simplest report generators I've seen in a long time, and the best thing is that it works. The operator prepares a Forms Definition file, again using a standard text editor, that defines exactly where on the printed page the data are to be put. This process is so easy that it's almost insulting. Titles, strings of characters to be inserted, and other formatting information go into this file, which is then used by VERSA-FILE to generate the final product. An interesting aspect of this report generator is that it can also be used to produce files of the data in your data set. These can be used as input to other programs such as MBASIC or WordStar/MailMerge, and vice versa. Very useful.

The problems I encountered were few, but one was major. The VERSA-FILE system relies heavily on full-screen formatting capability, and so must be modified for whatever cursor positioning protocol your system requires. The supplied installation program simply did not work. I have an H19, which is one of the terminals that is included on the installation menu, but I couldn't make the program believe me. I tried installing it as a custom terminal, and it still wouldn't take it. I finally did a file comparison on the kernal file, before and after installation, and there was no difference! I had to go in with ZSID, find the area where the cursor positioning strings were kept, and patch them. Since this program is evidently written in PL/1-80, this was not a trivial undertaking. My system is a standard CP/M 2.2, and I've never had this kind of problem before, so I think I'm justified in laying this at AMS's feet.

The second complaint I have about VERSA-FILE is minor. The program is menu-driven, so you switch modes a lot. At each step, the program asks you which datafile you wish to operate on. This is very annoying, and should be eliminated. A command such as dBASE II's USE, which specifies a default data set, would be a vast improvement.

All in all, VERSA-FILE is a remarkably powerful program, and should be considered by anyone needing a small data file management system. It's a no-frills package that can do a job reliably, and it has some bells and whistles that I've heretofore seen only in packages costing much more. For the price, you can't beat it. This is really an excellent example of the newest trend in the marketplace: good quality, low-cost software.

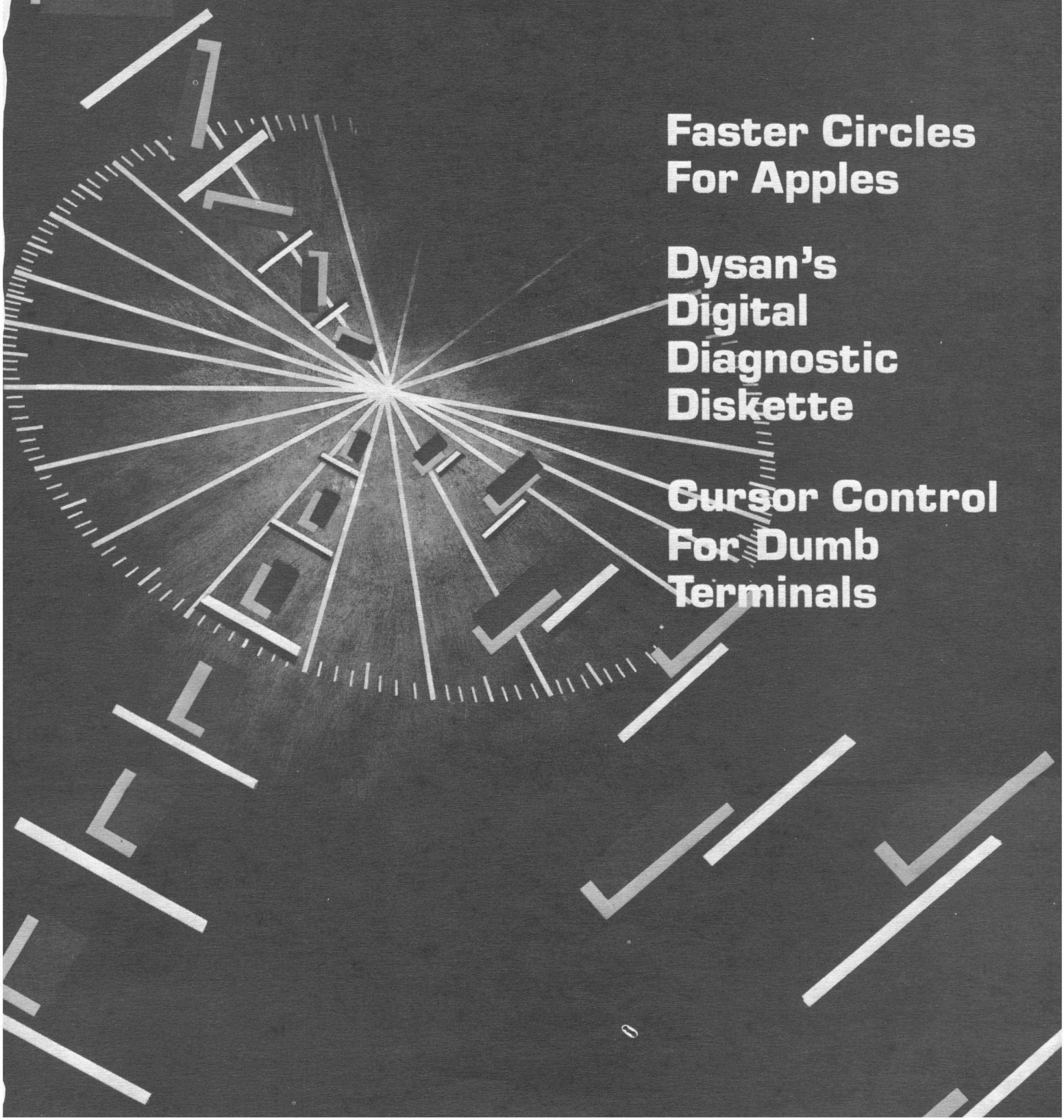
DDJ

#86 December 1983

$2.95 ($3.50 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems



Faster Circles For Apples

Dysan's Digital Diagnostic Diskette

Cursor Control For Dumb Terminals

Publisher – Jane Nissen Laidley
Editor – Reynold Wiggins
Managing Editor – Craig LaGrow
Contributing Editors –
Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg
Marketing Director – Beatrice Blatteis
Marketing Assistant – Sally Brenton
Marketing Coordinator – Ed Gueble
Advertising Director – Beatrice Blatteis
Advertising Sales –
Alice Hinton, Walter Andrzejewski
Circulation Director – Terri Pond
Circulation Assistant – Billie Greenwood
Production Manager – Barbara Ruzgerian
Production Assistants –
Shelley Rae Doeden, Alida Hinton, Tom Martin
Typesetter – Paula Fairchild
Cover – Alida Hinton

Copyright © 1983 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Christine Bell, W. D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, John W. Campbell, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Thomas Davis, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III, Infoworld, Stan Veit, Western Material Control, S. P. Kennedy, John Hatch, Richard Jorgensen, John Boak. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Homecomputer Vertriebs HMBH 282, Flugelstr. 47, 4000 Dusseldorf 1, West Germany; La Nacelle Bookstore, Procedure D'Abonnement 1-74, 2, Rue Campagne – Premiere, F-75014 Paris, France; Computercollectief, Amstel 312A, 1017 AP Amsterdam, Netherlands. **Asia & Australia:** ASCII Publishing, Inc., 4F Segawa Bldg. 5-2-2, Jingumae, Shibuya-Ku, Tokyo 150, Japan; Computer Services, P.O. Box 13, Clayfield QLD 4011, Australia; Computer Store, P.O. Box 31-261, 22B Milford Rd., Milford, Auckland 9, New Zealand. *(Write for Canadian distributors)*

Dr. Dobb's Journal
For Users of Small Computer Systems

December 1983
Volume 8, Issue 12

CONTENTS

ARTICLES

18 Faster Circles for Apples
by Myron L. Pulier
Inspired by Daniel Lee's "Fast Circle Routine" (May 1983 *DDJ*, No. 79), these circle-generation programs use Applesoft BASIC and assembly language to provide greater speed for Apple users. Two other *DDJ* readers also submit their amplifications on Lee's algorithm.

32 Cursor Control for Dumb Terminals
by Ian Ashdown
Want to teach your "antique" terminal to do direct cursor addressing? This author has developed a conversion program that's easy to implement on your CP/M system.

40 Dysan's Digital Diagnostic Diskette
by Loren Amelang
Tired of paying someone else a bundle of money to analyze and repair your faulty disk drive? Let this author show you how a program and Dysan's DDD can help you make your own diagnoses, without an oscilloscope or other fancy equipment.

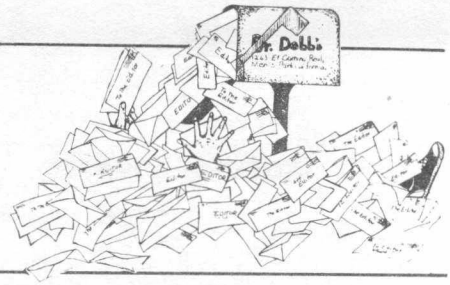
80 Building a Programmable Frequency Synthesizer
by Michael L. Simon
Here's how one author used a phase-locked loop to create a programmable frequency synthesizer. Using a spare counter/timer channel on a single-board computer, the configuration can multiply frequencies exactly.

DEPARTMENTS

9 Editorial
9 Letters
12 Dr. Dobb's Clinic
A brief look at two new computer columnists
14 C/Unix Programmer's Notebook
How Unix-type environments can lead to *non*-interactive and user-*un*friendly software
92 CP/M Exchange
Solutions to interfacing a hard disk within a CP/M environment, and how to speed up SUBMIT file truncation
102 16-Bit Software Toolbox
A look at the new MS-DOS EXEC function – a powerful "Unix-like" modification which should work well in a multi-tasking environment
112 Software Reviews
116 Book Reviews
120 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, P.O. Box E, Menlo Park, CA 94026. Second class postage paid at Menlo Park, California 94026 and additional entry points. Address correction requested. Postmaster: send form 3579 to People's Computer Company, Box E, Menlo Park, CA 94026 · 415/323-3111 **ISSN 0278-6508**

LETTERS



Augusta Q & A

Dear *DDJ*:

Edward Mitchell's four-part series on his Augusta compiler, concluded in the July 1983 issue, was certainly interesting. The sheer magnitude of effort that he put into it is mind-boggling. It was definitely a plus that he gave a detailed description of his program, rather than just a long listing.

But I find myself baffled by his penchant for doing things the hard way. He rightly points out that BASIC is a less than ideal language for compiler writing. OK, he couldn't find a version of Pascal, C, or Forth to run on his Osborne. (Really??) But since he's obviously interested in systems programming, why on earth use an Osborne?

Along the same lines, one wonders why he chose to use a recursive descent parser. When coded in a language that permits recursive subroutine calls, one can write such a parser simply, rapidly, and bug-free. But BASIC permits no such thing. Since he had to go to the trouble of maintaining a push-down stack to simulate recursion, why not write an augmented push-down machine parser? (See P. M. Lewis II, D. J. Rosenkrantz, and R. E. Stearns, *Compiler Design Theory*, Addison-Wesley, 1976.) Such an algorithm requires maintaining a stack of only the symbols in the same LL(1) grammar used for recursive descent parsing. Logically, the two algorithms are equivalent since each subroutine call in recursive descent corresponds to a symbol in the grammar. By doing "recursive descent" in BASIC, he is saddled with stacking subroutine calls *and local variables* – making his stack larger than necessary when memory is at a premium. In addition, the additional work needed to simulate recursion complicated his code and probably added enormously to the debugging time. A table-driven augmented push-down machine parser would have been shorter and simpler.

While I'm impressed by his having actually gotten Augusta up and running, it seems to be a classic example of an unnecessarily complicated solution to a problem resulting from choosing the wrong design and working with the wrong tools. I wouldn't recommend to others to follow that lead.

Very truly yours,
Clyde B. Schechter
520 West 122 Street
New York, NY 10027

Reply to Clyde Schechter:

Clyde Schechter raises some questions that other readers may also be curious about so I'll try to respond as best I can.

To put the Augusta project in proper perspective: Augusta was begun as a hobbyist project. That means it was done essentially for the sake of doing it, in the sense that rock climbers attempt difficult climbs solely for the sake of the climb.

Obviously, as I explained in Part IV of the Augusta series, neither BASIC nor an Osborne I was the ideal development vehicle for a large software project. But in view of the hobbyist nature of Augusta's development, purchasing a "real" computer was out of the question.

Languages other than BASIC were available for the Osborne I. But due to the 92K disk size on the Osborne I, the other languages were effectively useless. At least one of the languages required almost 140K for the compiler alone. That leaves little room for source code. And since the Augusta compiler had yet to be written, I could only estimate how much disk space I would need for source and object modules.

As I mulled over the choices, Microsoft BASIC seemed to be the best, a small program itself, with both interactive and compiled versions available. In addition, Tiny-Pascal, a project similar in scope to Augusta, had also been written in BASIC, so I knew that it was "do-able." In fact, the bulk of Augusta was written in about five weeks, working only part time. Once compiled with the BASIC compiler, Augusta was compiling at over 300 lines per minute, which compares quite favorably to other microcomputer compilers. So all in all, it really wasn't that difficult to write the compiler in BASIC. (The greatest difficulties were with the p-code interpreter which required several months of writing and tweaking.)

Mr. Schechter correctly observes that it's relatively easy to write a recursive descent parser in a language that permits

EDITORIAL

Here at *DDJ*, as we reach the end of 1983 and look back over the past twelve issues, we are encouraged by what we see. The number of pages is roughly twice what it was a year ago. Circulation has increased substantially. The new faces that have appeared in the columns have brought with them fresh enthusiasm and perspectives. The somewhat altered face of the magazine makes it a bit easier to find on the newsstands, and also a bit more readable. Our group of technical referees has been expanded, and authors are now being paid.

In the midst of all this change, we have fought to maintain our focus. Your responses to our recently completed reader survey have given us additional insight into what we are doing right and what we can improve, what you want and what you are doing. Our thanks to all those who responded.

Also helpful throughout the year has been the Editorial Response insert card. Please continue to use it to vote for your favorite item in the issue or to drop us short comments. The Doctor is never too busy to listen, and the postage is covered.

As we leave 1983, we look for 1984 to be one of the best years yet for *DDJ*. All of us here are rededicating ourselves to the task of improving the *Journal*. Our thanks to all of you for your continuing support. We wish you a happy holiday season, and look forward to serving you in the New Year.

Reynold Wiggins

recursive subroutine calls. Prior to writing Augusta I had worked on both a recursive descent parser for a BASIC interpreter (written in Pascal) and a table-driven LL parser for a subset of Pascal (written in Simula). Based on that experience, I found recursive descent to be easier to work with. In addition, table-driven predictive parsers are more difficult to implement in a one-pass compiler, and a one-pass compiler was an important design goal.

To sum up, his points are well taken. Undoubtedly it could have been done easier with a different set of tools. However, the overall development of the Augusta compiler was not as difficult as some may imagine it to be. Further, my plans all along have been to rewrite Augusta in itself. Having the original compiler as a model simplifies the rewrite, since the new compiler is also based on recursive descent parsing.

As of this writing, most of Augusta has been rewritten in Augusta. Unfortunately, I have been busy with other projects, including writing *Software Building Blocks for the IBM PC* (Hayden, 1984), as coauthor of Software Publishing Corporation's PFS:WRITE, and the teaching of a college summer school class. I will, however, be back working on Augusta in the near future.

I do appreciate everyone's comments, critical or praiseworthy. Thanks for writing.

Edward Mitchell

S-100 to Winchester Improved

Dear Editor,

David Cortesi has suggested some changes to my Winchester interface board (*Dr. Dobb's Journal*, October 1983) which would extend the range of CPU clock frequencies with which it would work. As an added benefit, the use and adjustment of the one-shots are eliminated.

Note that the modified schematic (Figure 1, below) includes a wait state generator, capable of generating one or two wait states, either of which may be selected by appropriately connecting the jumper. Detailed information about wait state generators may be found in Chapter 5 of *Interfacing to S/100-IEEE 696 Microcomputers* by Sol Libes and Mark Garetz, Osborne/McGraw-Hill, 1981. If additional wait states are needed, say for an 8 MHz microprocessor, then the D-type flip-flop could be replaced by a shift register. Details about the use of such a device are to be found in the same book.

Sincerely yours,
Oscar Goldman, Professor
Mathematics Department
University of Pennsylvania
Philadelphia, PA 19104

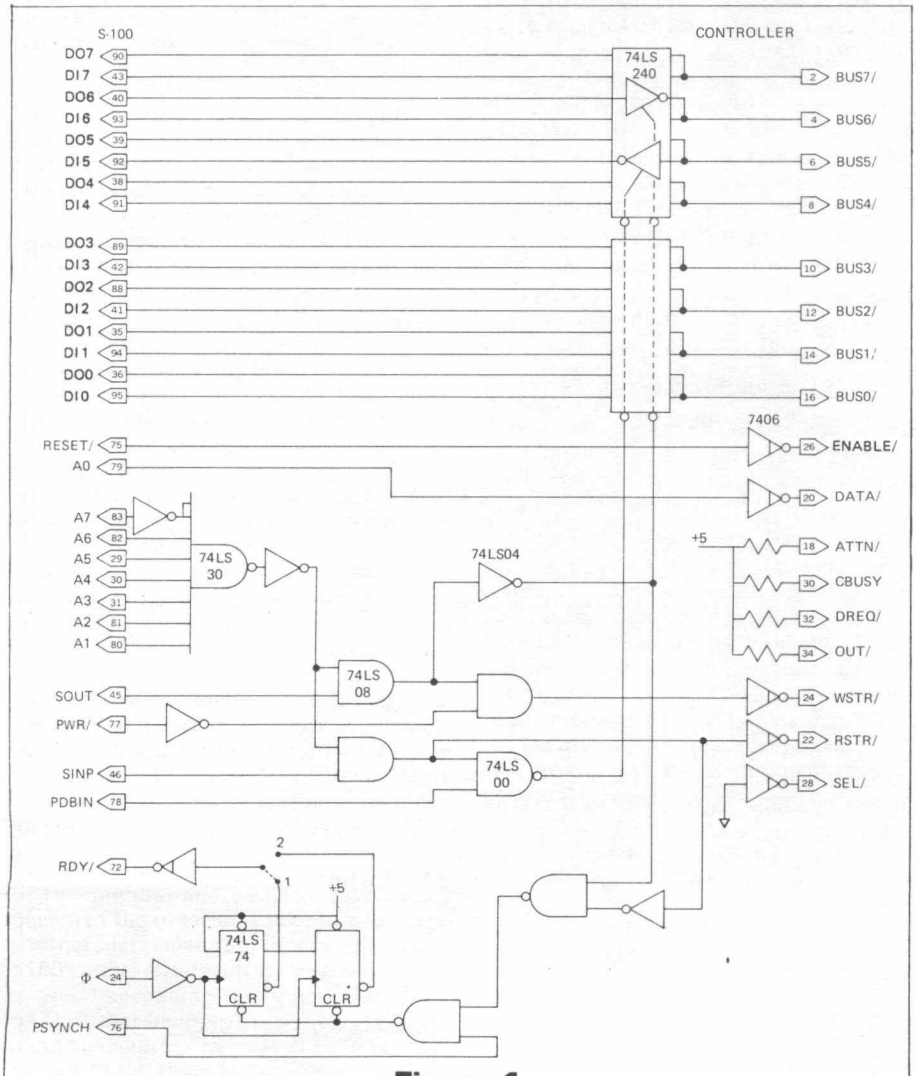


Figure 1.
The improved circuit diagram.

Recovering Lost WordStar Text

Dear *DDJ*,

Every now and then, for various reasons, I lose what I have just typed into WordStar. This induces a case of helpless rage. It's not the retyping I mind, it's the rethinking. I just can't face it.

The last time this happened to me, I didn't give up. WordStar was in an endless loop: beeping and inserting exclamation points in weird and what would otherwise be fascinating patterns. The keyboard was dead. Pulling out the printer interface card while the machine was running (not recommended!) stopped the malignant activity, but the keyboard was still dead. Eventually, there was nothing for it but CTRL-RESET. There went all my work...

But, wait. Maybe it's still in memory! I called up DDT and used the D command to examine all of memory, from the beginning. Sure enough, at hex 7849 on, there were my precious words! I used the DDT M command to move my text to 0100. Back to the system with CTRL-C. Save the recovered material (e.g., SAVE 6 HOPE). Not only can it be typed (TYPE HOPE), but I could use it as input to WordStar with no problems! (Well, there was garbage at the end, a small price to pay, and no doubt inserting a proper terminator with DDT would have cured that.)

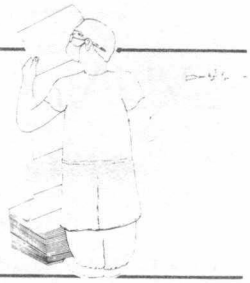
Too late to recover that three-page letter I wrote a month ago – but I'm ready for next time. Don't wait until it happens to you. Practice now. Be prepared!

Keith K. Davison
239 Clinton Road
Brookline, MA 02146

More Circles and Such

We received several items amplifying the circle generation discussion begun by Daniel Lee in the May 1983 DDJ. *In order to present them all together, this month the Letters column spills over to the sidebars on pages 14-16. – Ed.*

DR. DOBB'S CLINIC



by D. E. Cortesi, Resident Intern

Readers of this column should know about two recent startups in the column industry. The first is a big-time effort that, we think, faces grave difficulties. The second promises to be a gem of great value and deserves to be widely read.

Big-Time Columnizing

Many of you will already know that Douglas Hofstader has retired his "Metamagical Themas" in *Scientific American* and that Brian Hayes has stepped into that hallowed space (originally occupied by Martin Gardner's "Mathematical Games") with "Computer Recreations." His first effort, a consideration of the fundamental shift in thinking encouraged by spreadsheet calculators, appeared in the October issue. It's a welcome sign when the august *SciAm* recognizes the prevalence, and the fun, of personal computing, and we wish Hayes the best of luck.

System-Dependent Prose

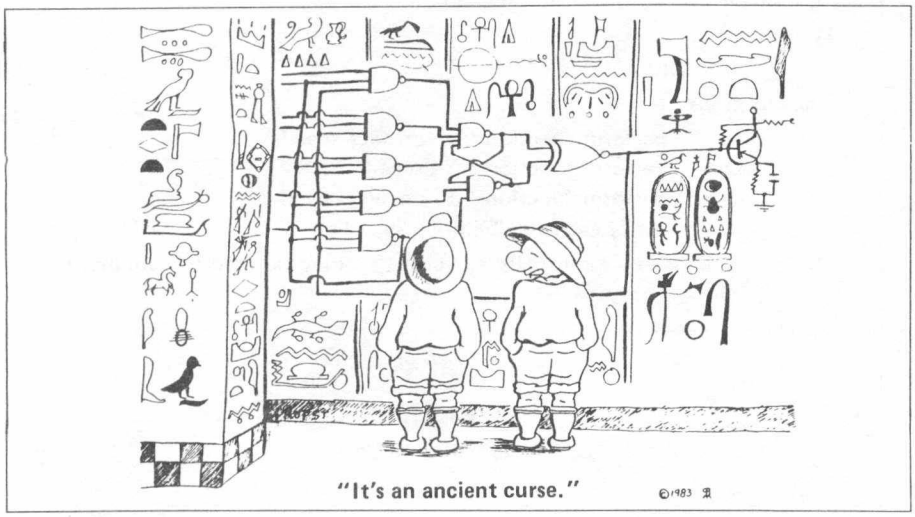
However, columnizing isn't easy in the best of circumstances, and he faces some difficulties. We wonder how (given the magazine's huge circulation, the diversity of its readers' interests, and the diversity of the computers they use) he is going to proceed. Spreadsheets are safe enough for a first outing; everybody has one. But when it comes time to present algorithms, what language will he use? When it is time to draw a graph, can he avoid hardware dependency?

The infernal, niggling differences between one language (machine, keyboard, disk drive, etc. *ad nauseum*) are the curse of computer publishing, and nobody has found a decent way to produce system-independent prose. It isn't a new problem, either. The Association for Computing Machinery (ACM) faced it a long time ago and decided to publish algorithms in "presentation ALGOL," the ALGOL syntax augmented with a complicated set of typographical conventions. Donald Knuth faced it and opted to design his own CPU, MIX, and to present algorithms in its assembly language, MIXAL. These solutions are essentially the same. They proceed from the assumptions that (1) no two readers use quite the same language, and (2) each reader knows some language and system very well. Therefore all readers will be presented with a language that *nobody* knows and will all have to learn it; afterward they can translate to their own environment.

The personal-computer press usually operates on the theory that everybody knows and uses some form of BASIC, and that all readers are good enough programmers to convert from one BASIC's idiosyncracies to those of their own machine's BASIC. It isn't true, of course. We once published a program for the IBM PC (in another magazine) and later received a long distance call from an unhappy reader who simply couldn't understand why the program didn't work on his Victor 9000.

Another solution is to embrace the problem and publish system-dependent material only; hence we have magazines for the Apple, the Atari, the Color Computer, the IBM PC; for CP/M; for the Heathkit product line; for machines that use the 6502. And we have books that appear in multiple editions with only trifling changes in their content, like the series of *The Power of XXXXCalc* books. This isn't a solution at all, of course; it only hides the problem. It drives up costs, fragments the audiences, and prevents the dissemination of good solutions to many users.

So how will Hayes handle the problem? Can he find a way to present problems and solutions to a mass audience in a way that will be interesting and useful to most of his readers? We'll see.



"It's an ancient curse." ©1983 JR

Bently's Pearls

Communications of the ACM, longtime flagship of computer science journals, launched a new column in its August issue. The columnist is Jon Bently, an Associate Professor at Carnegie-Mellon and author of *Writing Efficient Programs* (a book that, on the evidence of Bently's debut column, we certainly plan to buy).

The column's title is "Programming Pearls," the pearls being those parts of programs whose origination require "insight and craftsmanship." The pearls presented in the first column were delightful and gave us several hours of vigorous, enjoyable mental exercise. Bently has avoided the problem of system dependency by keeping his problems abstract and by presenting algorithms in English and, where necessary, in a simple pseudo-code.

Bently plans to end each column with questions and problems, but he isn't asking for solutions from his readers (from our experience, we'd call that a mistake; the massed mental power of a motivated readership is awesome). Here is one of the problems he presented.

"Rotate a vector of N elements by I positions. For instance, with $N=8$ and $I=3$, the vector *ABCDEFGH* is rotated to be *DEFGHABC*. It is easy to do this rotation in N steps if we have available an N-element intermediate vector. Can you rotate the vector in time proportional to N using only a few extra words of storage?"

We could; we ended up using five scratch variables. Furthermore, when we expressed our solution as a BASIC subroutine (taking variables I and N and vector V as input, modifying the contents of V), we found that with a single added line of code we could accept a negative value of I to mean "rotate right." Can you find a similar pearl for this oyster of Bently's? If you enjoy searching for that one, be sure to read the rest of his column. ■■J

C/UNIX PROGRAMMER'S NOTEBOOK

by Anthony Skjellum

In the last column, several references to the book *The C Programming Language* and its authors were made. Through my error, Brian Kernighan's name was misspelled consistently throughout the column. I'm sure that many readers noticed this immediately. Unfortunately, I didn't until I saw the column in print. [*We obviously missed it too. Our apologies. –Ed.*]

In this second Programmer's Notebook, I'll discuss how Unix-type environments can lead to non-interactive, and user-unfriendly, software. This is based on experiences I've had with several Unix and Unix-like systems running both standard and Berkeley Unix.

Skimming through the ads of the October *DDJ*, I noticed a very interesting item: the Computer Innovations ad concerning the soon-to-be-released version of their C compiler. As an option, this compiler will produce programs/data exceeding the previous limit of 64K segments imposed by all 8086 compilers (C or otherwise) that I've seen. Look for a review in this column early next year.

Unix Software

The Unix operating system was designed to reduce repetition of programming effort by permitting modular programs to be combined via pipes and tees. Since input and output are redirectable under Unix, simple programs could use console input and output for one application and be used as part of a pipeline for another. Thus, unmodified programs could be reharnessed for new applications to an extent not possible with previous operating systems.

Pipes and input-output redirection are two of the best and most well-known features of the Unix system. Microcomputer users have been very interested in adding these capabilities to their own operating environments. In the 8-bit world, this has been done chiefly through special subroutine libraries such as "The Unica," or in C runtime packages. For MS-DOS 2.0 users, the features are built into the operating system.

Despite the undisputed usefulness of pipelines and input-output redirection, their presence in Unix has lead to a serious drawback in the system's environment. This drawback is the proclivity to avoid interactive programs and to produce user-unfriendly software. Furthermore, the standard Unix console interface is weaker than under other operating systems. In the remainder of this column I will discuss

1. Terse (hard-to-remember) program names.
2. Lack of program sign-on and sign-off messages.
3. Lack of interactive mode to alleviate the need to re-execute a program several times to complete a set of operations.
4. Inconsistent use of switch (dash options) for controlling the specifics of program execution.
5. Lack of descriptive error messages.
6. Cryptic, incomplete, and erroneous documentation.
7. Software bugs: undocumented and documented.
8. Cryptic (or missing) internal help features.
9. Poor console interface provided by Unix.
10. Lack of system for finding program names by the function required.

Table 1
Unix Software Problem Areas

```
$ fct <RETURN>              (activate the program with no arguments)
usage: fct -abcv file1 . . fileN   (help message)
$                           (shell prompt)
```

Figure 1.
Cryptic fct Session

```
$ fct <RETURN>              (activate the program with no arguments)
fct (version x.yz) as of dd-mm-yy
usage:
        fct [dash options] file1 . . fileN
dash options:
        -a      perform function "a" on files specified
        -b      perform function "b" on files specified
        -c      perform function "c" on files specified
        -v      verify each step before proceeding
note:   -a and -b are mutually exclusive; -c may be used in conjunction
        with -b only.
End of execution.
$                           (shell prompt)
```

Figure 2.
Friendly fct Session

these weaknesses as I perceive them.

Non-Interactive Software

Because of the availability of pipes and input-output redirection, many Unix programs are designed to act as filters. Filters are programs which require a single sequential input data stream and produce a single output data stream. Such programs are suitable as pipe-fittings. Because of the way they handle data, they don't normally expect to be used interactively. In most cases this doesn't pose a problem for users. However, because such programs do not expect to deal directly with humans, but only with input and output streams, they can often be very unfriendly in handling errors.

The problem in the Unix operating system is that the same terse philosophy applied to filters also pervades most of the software available. This includes programs which are normally executed sequentially by the user from the console. The problems come in several areas and some of these problem areas are listed in Table I (page 14). The following paragraphs will elaborate each of the points listed in that table.

Terse Names

Unix program names are usually two or three letters long and tend to be cryptic. While this saves typing for experienced users, it's frustrating for new and occasional system users. Also, since the Unix system lacks an on-line indexing system for finding program names by function, it's not easy to find the right program based on the desired function alone.

Sign-ons and Sign-offs

Sign-on and sign-off messages are a common courtesy in the computer world. Virtually all standard Unix programs lack these two simple features. While this is understandable for filters, it is completely unnecessary for other programs. For example, if two versions of a program exist on a system, the only easy way to distinguish them is by their sign-on messages.

Besides sign-ons/sign-offs, it is also nice for a program to give progress reports during execution. This lets the user know how things are proceeding. Standard Unix software doesn't normally include such a feature.

Internal Help Features

Many programs include a feature summary option to help occasional and new users remind themselves of program operation. Many Unix programs also have this capability, but they are often extremely cryptic and include few English words to supplement the sample command line which they display. Figure 1 (page 14) displays a sample session in which a fictitious program, **fct**, is executed from the shell with no arguments. The program responds with a cryptic help summary typical of actual Unix commands. In Figure 2 (page 14) the same **fct** program session is presented, but this time the program has been designed to provide a user-friendly help feature (and also to sign on and off).

Interactive Modes

Interactive program modes provide a friendly environment for the user. When a program is used often, it may be executed several times consecutively. An interactive mode eliminates the need for consecutive execution since the user can enter all the commands in one interactive session. This avoids unnecessary user effort and is probably more efficient from a system standpoint.

One reason that the interactive modes are missing is the lack of support for expanding Unix wildcard filenames from within a program. I find this limitation rather arbitrary, and I have written a program which solves the problem (see "Expanding Unix Wildcards," *DDJ*, No. 73, November 1982). The lack of direct support for such a function indicates that the whole Unix philosophy is geared towards non-interactive software tools.

Command-line Switches

Unix programs use command-line switches (dash options) to specify the particulars of program execution. Switches usually consist of a dash character ("-") followed by a single letter. Unfortunately, Unix programs use these switches inconsistently. For example, the **rm** command allows only limited positioning of switches, permitting them only before filenames on the command line. Some programs allow several switches to be combined after a single dash character (e.g., -abcdef). Others use the plus ("+") character to activate a program feature and a dash to deactivate it. Generally, it is difficult to remember all the different possibilities, limitations, and defaults imposed by the various programs.

The difficulties with command-line switches under Unix led me to write the ARGUM package which was published in the August 1982 issue of *DDJ* (No. 70). That article proposed a program which would handle switches in a consistent way. Existing Unix programs could be changed to use ARGUM, thus eliminating one degree of inconsistency from the operating environment. Alternatively, the less powerful Unix III **getopts()** facility could be used.

Lack of Descriptive Error Messages

The lack of descriptive error messages is a real problem, especially for inexperienced users. One offender is the **eqn/troff** system used for equation and text photo-typesetting. These programs report errors as "Syntax Error" between two line numbers. They don't echo the erroneous text or equation, and the line numbers aren't always useful because header files change the length of the source text.

Another problem stems from the way Unix reports failures. Programs which attempt to open a file and fail get an error code. They subsequently report the failure as "cannot open file." While this is correct, it doesn't tell the user if the file is nonexistent or if a file protection violation has occurred. Similarly, when users try to change their current directory to one for which they have no read privileges, a "bad directory" message is displayed by the Unix shell.

Documentation

Documentation is a real problem in the Unix system. Most programs are documented in a standard form which is typically very terse. The examples provided are often very complicated and don't clearly illustrate how to use the program for simple purposes. Because of the poor documentation, some casual users think of Unix systems as secret societies since only the indoctrinated can tell them how the system and programs work.

Another problem with documentation is that it sometimes doesn't reflect the current state of a program. Most notable on the system I use are undocumented features of the standard Unix editor **ed**. The standard form provides no line edit command ("x") while the installed version does. Yet, the documentation does not explain this.

Unix documentation is clearly the weak link in the system. It makes some of the mysterious concepts of the system seem impossible to grasp and reduces productivity through its terseness.

Software Bugs

Another problem which makes a Unix system user-unfriendly (actually user-hostile) is the presence of undocumented bugs in important software packages. For example, bugs exist in the **eqn/troff** system. These can be circumvented, but the methods are known only to a few experts. No generally available documentation exists for avoiding such problems.

Since the Unix system comes with source code, it should be feasible for individual users to change system software to their specific needs. Unfortunately, the source code which accompanies Unix is mostly comment-free and is therefore difficult to understand without significant effort.

User Console Interface

The user console interface is not extremely good under Unix. For example, when a character is deleted, Unix does *not* actually remove the character typed but just moves the cursor back one space. There is no standard mechanism for having a line retyped, and backspacing after typing a tab doesn't produce the correct results. Furthermore, when a control character is typed, it is displayed in the form ^CHAR, but when deleted, the cursor moves back only one space (leaving the caret).

The weak user interface points again to the philosophy that most software will not be interactive. However, the standard Unix editor makes no attempt to improve the interface for the purpose of interactive editing. It is so unfriendly that most users resort to other editors (usually screen editors).

Programmers can only provide their programs with a superior console interface by using the raw terminal mode. Unfortunately, this mode is more expensive in terms of input-output cost. For user-friendly, screen-oriented software, it's the only way to go.

Conclusion

Unix is a powerful operating system, and is certainly one that I enjoy using. It does, however, have a number of user-unfriendly aspects, and the system philosophy has led to a predilection for non-interactive software.

I look forward to hearing how other Unix users perceive the Unix environment.

■■J

Faster Circles for Apples

by Myron L. Pulier

Myron L. Pulier, M.D., 101 Cedar Lane, Teaneck, New Jersey 07666.

Daniel Lee's article, "Fast Circle Routine," in *DDJ* No. 79 (May 1983) inspired me to create a similar circle-making approach for the Apple. Although I first wrote an Applesoft BASIC program to test the logic, I chose variable names that could serve for both BASIC and assembly language. Thus "X" and "Y" refer to the X and Y registers of the Apple's 6502 chip, and "Xcrd" and "Ycrd" are the coordinates for the points of the circle.

A circle of radius R centered at (Xmid, Ymid) is described by the familiar formula

```
(Ycrd-Ymid)^2
    + (Xcrd-Xmid)^2 = R^2
```

where Xcrd is the horizontal variable, Ycrd is the vertical variable, and Xmid and Ymid are constants. Differentiating with respect to Xcrd gives

```
2*(Ycrd-Ymid)*dYcrd/dXcrd
    + 2*(Xcrd-Xmid) = 0
```

whence

```
dYcrd/dXcrd =
    - (Xmid-Xcrd)/(Ymid-Ycrd)
```

The last equation implies that, in drawing the circle, if we increase Xcrd by 1 to plot the next point we must decrease Ycrd by

```
(Xmid-Xcrd)/(Ymid-Ycrd)
```

The slowest operation here is division by Ymid-Ycrd, which must be performed each time we want a new value for Ycrd. We can reduce the number of these divisions by evaluating the expression for only one eighth of the circle and by plotting the rest of the circle symmetrically about the coordinate axes and about a diagonal.

It is best to select the upper left extreme of the circle as the starting point. According to the Apple coordinate system, where point (0,0) is the upper left corner of the screen, our starting point is given by

```
(Xmid-R/SQR(2),Ymid-R/SQR(2))
```

From here we move to the right and stop at the extreme top of the circle, which is point (Xmid,Ymid-R). This choice of starting and ending points facilitates a simple FOR-NEXT program loop (FOR Xcrd = Xmid-R/SQR(2) to Xmid) and avoids the divide-by-zero error we might encounter at the extreme right and left of the circle, where the slope is undefined.

Reader Commentary

More Fast Circles . . .

Dear *DDJ*,

Daniel L. Lee's algorithm has got to be faster than Microsoft's pedestrian CIRCLE command, but both suffer from the same malady: they reinvent the wheel – only this time it's square!

When I think I've discovered a marvelous algorithm, I wonder if I've outsmarted the professionals. I usually haven't. But hope springs eternal. I search the literature anyway. My brainchild is at least 17 years old [B. K. P. Horn, "Circle Generators for Display Devices," *Computer Graphics and Image Processing* (5), pp. 280-288 (1976)].

Neither trigonometry nor calculus is needed to devise a circle generator. For a circle of radius R one wants to plot points (X,Y) with integer coordinates which most nearly solve the equation

$$X^2 + Y^2 = R^2$$

The difference between the left side and R^2 is a measure of nearness. A suitable circle generator simply chooses successive points to minimize this difference. The enclosed listing (see Listing Three, page 30) is a rendering of such an algorithm. It generates points for about one eighth of the circle and, using the symmetry of the circle, plots eight points for each point generated. For use with digital plotters, the algorithm is invoked eight times forward and backward so that the points are of concentric circles; the low algorithm is plotted in counter-clockwise order. I enclose a plot of concentric circles in low resolution to exhibit the algorithm's behavior (see Figure 2, below).

William A. McWorter, Jr.
Mathematics Department
Ohio State University
231 W. 18th Avenue
Columbus, OH 43210

(Listing Three begins on page 30)

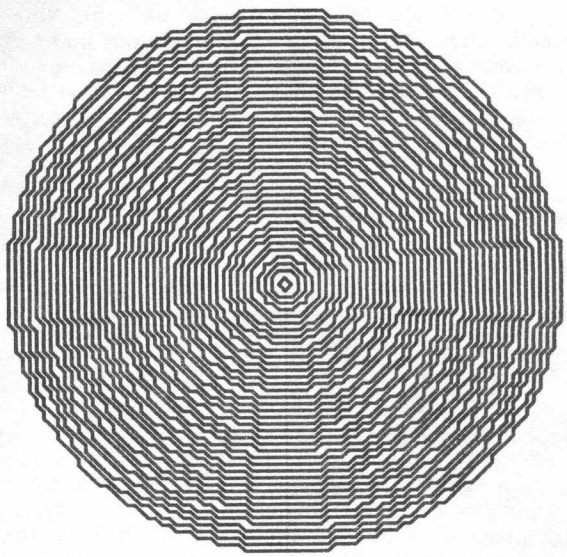


Figure 2.

See Figure 1 (at right).

For each point that we find by the above method, we can generate seven symmetric points by reflection through the horizontal and vertical diameters of the circle and by exchanging Xcrd with Ycrd. A point (Xcrd,Ycrd) is ABS(Xcrd-Xmid) distant from the vertical axis of the circle. Since its reflection should be the same distance from this axis, its abscissa is 2*Xmid-Xcrd. Similarly, reflection through the horizontal diameter gives an ordinate of 2*Ymid-Ycrd. To reflect a point through the diagonal line that runs from upper left to lower right, we find the point whose distance to the vertical axis equals the distance of the original point to the horizontal axis and whose distance to the horizontal axis is the same as the original point's distance from the vertical axis, namely

(Xmid+Ymid-Ycrd,Xmid+Ymid-Xcrd)

This new point can now be reflected as before through the vertical and horizontal axes.

In Listing One (page 21), subroutine 9000 plots four points symmetrically about the horizontal and vertical axes. Lines 10050 and 10055 switch the X and Y coordinates, then line 10060 calls 9000 to plot the four new points. The parame-

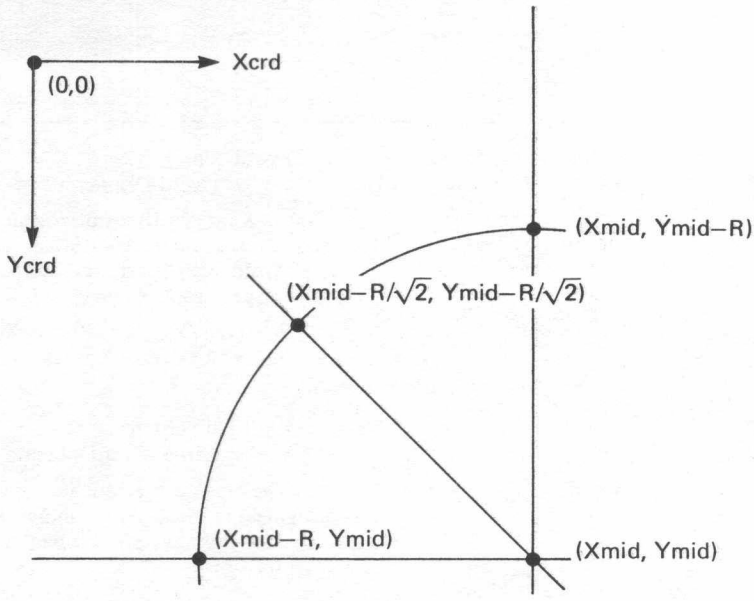


Figure 1.
Faster Circle

Reader Commentary

. . . And Fast Ellipses

Dear Sirs:

Mr. Daniel Lee, in the May '83 issue, presented a fast circle generator. It compensated for any given screen aspect ratio, and as such may be used as an ellipse generator. I submit the algorithm described below as an even faster alternative. The speed improvement results from the elimination of all division and most of the multiplication. The approach taken could easily be modified to allow the generation of arcs.

The method which I present here is based on the equation of the circle, and a trick which eliminates a great deal of multiplication. There is no calculus or trigonometry involved, implicitly or explictly.

The equation of the circle is well known:

$$x^2 + y^2 = r^2 \qquad [1]$$

where r is the radius. Since we want to minimize multiplication, we have to use "magic." A magical property of the positive integers is that the square of a positive integer *n* is the sum of the first *n* odd numbers. This means that if we want to compute x^2 for each x we can actually plot (i.e., each integer x), we only need to know which odd numbers to add up. The same applies to y^2.

In order to plot a circle, we might start at the point (0,r) and plot towards (r,0), using symmetry to generate the other arcs of the circle. This would mean that x would go from 0 to r, y would go from r to 0, x^2 would go from 0 to r^2, and y^2 would go from r^2 to 0. It is easier than it first appears to calculate y^2. Note that y^2 is the sum of the odd numbers from 1 to 2y-1. In the initialization phase it will be necessary (perhaps) to compute y^2 directly, but for y' = y-1, $y'^2 = y^2 - (2y-1)$.

Above I said "perhaps" because it develops that one does not need to refer directly to y^2 or even to x^2. The procedure for drawing the circle requires that we assume, as we did above, that we will draw primarily from (r,0) to (0,r) and use symmetry to generate the rest of the points. As we compute the points for the primary arc, we maintain a total *e*. The total starts at 0; for every time we actually move in the positive x direction, we add 2x-1 to *e*; for every time we actually move in the negative y direction, we subtract 2y-1 from *e*. We decide precisely which step or combination of steps to take by insisting that the *e* that would result from the step or combination of steps be as close to 0 as possible.

An Ellipse

To generate an ellipse is a slightly more complex matter, but in the end we lose little speed. The equation for an ellipse centered at the origin is

$$b^2x^2 + a^2y^2 = a^2b^2 \qquad [2]$$

where b is the positive y-intercept, a is the positive x-intercept, and a/b is the resulting aspect ratio. I claim that in order to successfully trace the ellipse we need only do exactly as we do for the circle, but we must multiply every reference to x by b^2 and every reference to y by a^2. In other words, every time we actually move in the positive x direction, we add $b^2(2x-1)$ to *e*; for every time we actually move in the negative y direction, we must subtract $a^2(2y-1)$ from *e*. Again we decide which step or combination of steps to take by insisting that the *e* that would result from the step or combination of steps be as close to 0 as possible. In this case we are plotting from (0,b) to (a,0).

If perhaps the terms $b^2(2x-1)$ and $a^2(2y-1)$ look like they involve too much multiplication, please realize that in fact no multiplication is required. For example, we would already know the evaluation of $b^2(2x-1)$ to

(Continued in box on page 20)

ters R2, X2, Y2, and XY have been introduced to speed computation.

In the segment of the circle from the upper left point through the upper middle, the change in Ycrd is fractional for each unit change in Xcrd. Because the Apple plotting routine deals with integers, the decrement in Ycrd builds until it causes the line being drawn to move up one full position.

The assembly program in Listing Two (page 22) runs much faster than its Applesoft equivalent. Since Xcrd can range from 0 through 279, it must be a double-precision variable. It occupies locations XCRDH and XCRDL. Ycrd is supplemented by a fractional portion stored in YCRDF. Names of other double-precision integer parameters are terminated with -H or -L for the high- and low-order portions, respectively. Single-precision assignment is indicated in the comments by "<—", while double precision is "<<—".

The TEST program plots a circle of radius 40 and midpoint (120,80). It initializes the hires screen by calling TURNON. The subroutine called EIGHTH performs calculations for the one-eighth circle. Here the first order of business is to approximate the value of R/SQR(2) by using R*3/4 instead. Note that 3/4 in decimal is 1/2 + 1/4, or 0.11 in binary.

The next lines of the assembly program are a straightforward translation of their Applesoft equivalents. Lines 75 and 76 initialize the value of YCRDF to 0. PLOTFOUR is called in lines 104 and 119 to place four points symmetrically about the horizontal and vertical axes of the circle. PLOTFOUR uses the Applesoft HPLOT routine to perform the actual plotting. HPLOT requires that the horizontal coordinate be in the Y and X registers,

(Continued from page 19)

be, say, e_x. To determine $e_{x'}$ when $x' = x+1$, note that

$$b^2(2x'-1) = b^2[2(x+1)-1] = b^2(2x-1)+2b^2;$$

in other words,

$$e_{x'} = e_x + 2b^2.$$

A similar result obtains for the negative y direction, which we will simply state:

$$e_{y'} = e_y - 2a^2.$$

Algorithm Summary

To summarize the algorithm: start with the point (0,b). Initialize *e* to 0, e_x to b^2, e_y to $2a^2b-a^2$, e_{xy} to e_x+e_y. Plot the current point and corresponding points in the other quadrants of the ellipse. Choose the next point so that *e* plus $e_{whatever}$ is minimized. Set *e* according to that choice, and update e_x, e_y, and e_{xy}. When the point (a,0) is arrived at, the ellipse is complete.

The Listing

The program shown in Listing Four (page 30) is an MBASIC program intended to interface to an LSI ADM-3A terminal. Obviously, if speed is a concern, BASIC is not the language of choice. I chose it to permit the program to be tried out basically anywhere, since my facilities for computer graphics are one-of-a-kind.

Lines 1050-1240 are the routine itself. The point-plotting routine is on lines 1310-1341.

Caveat

There is one thing that the implementor should be aware of before he or she starts, to prevent untraceable bugs. The formulae for e_x, e_y, and e_{xy} include squares of a and b. These squares accumulate to a large total rather quickly. The solution is to use a wide word to store the total, and perhaps (depending on the size of your screen in pixels) the values of e_x, e_y, and e_{xy} as well.

Drawing Arcs

The method can be modified to draw arcs (see Figure 3, below) elliptical or otherwise, with careful initialization and a well-considered termination condition. The initialization involves calculating e_x, e_y, and e_{xy} for the initial point of the arc to be drawn. The routine should terminate when the last point of the arc is drawn. The actual coordinates of the final point should be calculated in some fashion that allows for rational numbers, and then a point with integer coordinates should be chosen that approximates the actual point. This can be done by using the equation of the ellipse. In other words, the best integer approximation (x_i,y_i) of the terminating point (x,y) is the one for which $(bx_i)^2+(ay_i)^2$ is closest to $(ab)^2$. Again, the integer coordinates of the final point should be computed in the initialization phase and used as the termination condition.

Conclusion

This routine can draw an ellipse quickly, using no multiplication once initialized. It should be easily implemented in 68000 assembly language, owing to that processor's 32-bit register operations. A little more difficulty should be anticipated by users of the 8086, 6809 or Z80, though their 16-bit addition capabilities can be used to advantage. HLLs can speedily draw circles with this routine, as well, because of its incremental nature. And finally, the algorithm can draw arcs easily.

Michael T. Enright
2360 Hosp Way, #132
Carlsbad, CA 92008

(Listing Four begins on page 30)

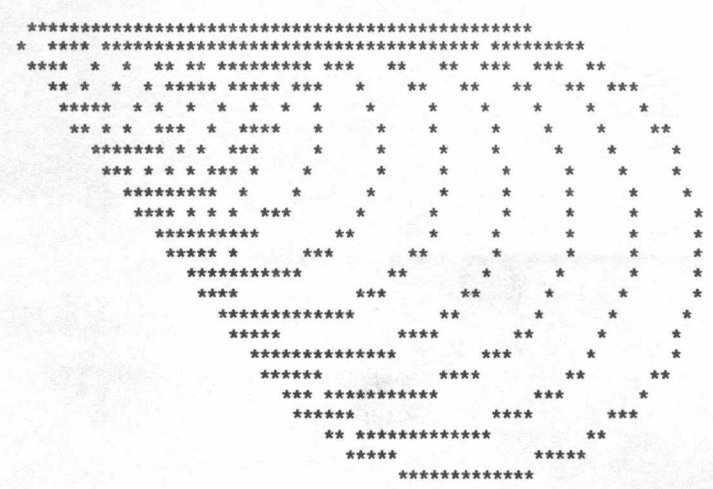


Figure 3.

and the vertical coordinate in the A register. HPLOT returns these coordinate values at the zero-page locations named LASTHH, LASTHL, and LASTV.

The division sequence starts on line 135. Since the divisor and dividend are single precision, we can use a technique that divides a one-byte divisor stored in DIVISOR into a one-byte dividend held in the A register. The eight shift operations and eight subtractions required are counted via the X register. The result of the division is a binary fraction generated in QUOTIENT. This quotient is subtracted from the previous value of YCRDF. If a borrow is required, we decrement the integer portion of Ycrd. In any case, Xcrd must be incremented by 1 in a double-precision operation. The FOR-NEXT loop of the BASIC version is implemented in assembly language by counting the value of R2 down through zero, since R/SQR(2) points will be plotted for one eighth of the circle.

Using zero-page locations for variables and parameters and a faster division algorithm will increase speed, but the bottleneck is the Applesoft HPLOT routine, which maps horizontal and vertical coordinates into the Apple video locations. Replacing that routine with a table lookup results in very fast circle generation.

With some modification the "faster circle" technique can produce filled-in disks or wedges for pie charts. It can also rotate and translate shapes and objects quickly for animation effects.

(Listings begin below)

Circles (Text begins on page 18)

Listing One

```
1000    ***************TURNON

1005  Hgr
1010  Hcolor = 3


2000    ***************TEST

2005  R = 40
2010  XMID = 120
2015  YMID = 80
2020  Gosub 10000
2025  End


9000    ***************PLOTFOUR

9005  Hplot H,V
9010  Hplot X2 - H,V
9015  Hplot X2 - H,Y2 - V
9020  Hplot H,Y2 - V
9025  Return


10000   ***************EIGHTH

10005  R2 = R * .7
10010  YCRD = YMID - R2
10015  X2 = XMID + XMID
10020  Y2 = YMID + YMID
10025  XY = XMID + YMID
10030  For XCRD = XMID - R2 To XMID
10035       H = XCRD
10040       V = YCRD
10045       Gosub 9000
                  * PLOT 4 POINTS
10050       V = XY - XCRD
10055       H = XY - YCRD
10060       Gosub 9000
                  * PLOT 4 POINTS

10065       YCRD = YCRD -
                  (XMID - XCRD) /
                        (YMID - YCRD)
10070  Next
10075  Return

END OF LISTING

PROGRAM LENGTH = 421 BYTES,
          TOTAL OF 31 LINE NUMBERS


27 TOTAL NON-REM STATEMENTS,
                    6 TOTAL REMARKS


END
PR#0
```

End Listing One

Listing Two

```
0800                    3           LST
0800 2C 50 C0           4  TURNON   BIT GRAPHICS
0803 2C 52 C0           5           BIT FULL
0806 2C 54 C0           6           BIT PAGE1
0809 2C 57 C0           7           BIT HIRES
080C A9 FF              8           LDA #$FF
080E 85 E4              9           STA HCOLOR
0810 A9 20             10           LDA #$20
0812 85 E6             11           STA HPAGE
0814                   12  ;
0814 A9 28             13  TEST     LDA #40             ;R<-40
0816 8D 2F 09          14           STA R
0819                   15  ;
0819 A9 78             16           LDA #120            ;XMID<<-120
081B 8D 37 09          17           STA XMIDL
081E A9 00             18           LDA #0
0820 8D 36 09          19           STA XMIDH
0823                   20  ;
0823 A9 50             21           LDA #80             ;YMID<-80
0825 8D 3C 09          22           STA YMID
0828                   23  ;
0828 20 64 08          24           JSR EIGHTH          ;DRAW CIRCLE
082B 60                25           RTS
082C                   26  ;
082C AC 2D 09          27  PLOTFOUR LDY HH              ;HPLOT H,V
082F AE 2E 09          28           LDX HL
0832 AD 31 09          29           LDA V
0835 20 57 F4          30           JSR HPLOT
0838 AD 33 09          31           LDA X2L             ;HPLOT X2-H,V
083B 38                32           SEC
083C E5 E0             33           SBC LASTHL
083E AA                34           TAX
083F AD 32 09          35           LDA X2H
0842 E5 E1             36           SBC LASTHH
0844 A8                37           TAY
0845 AD 31 09          38           LDA V
0848 20 57 F4          39           JSR HPLOT
084B A4 E1             40           LDY LASTHH          ;HPLOT H2-H,Y2-V
084D A6 E0             41           LDX LASTHL
084F AD 3B 09          42           LDA Y2L
0852 38                43           SEC
0853 E5 E2             44           SBC LASTV
0855 20 57 F4          45           JSR HPLOT
0858 AC 2D 09          46           LDY HH
085B AE 2E 09          47           LDX HL
085E A5 E2             48           LDA LASTV
0860 20 57 F4          49           JSR HPLOT
0863 60                50           RTS
0864                   51  ;
0864 AD 2F 09          52  EIGHTH   LDA R               ;R2<-R*3/4
0867 4A                53           LSR
0868 18                54           CLC
0869 6D 2F 09          55           ADC R
```

(Continued on page 24)

Circles (Listing continued, text begins on page 18)

Listing Two

```
086C 6A          56              ROR
086D 8D 30 09    57              STA R2
0870             58  ;
0870 AD 3C 09    59              LDA YMID            ;YCRD<-[YMID-R2]
0873 AA          60              TAX                 ;X<-YMID FOR LATER
0874 38          61              SEC
0875 ED 30 09    62              SBC R2
0878 8D 3D 09    63              STA YCRD
087B             64  ;
087B AD 37 09    65              LDA XMIDL           ;X2<<-[2*XMID]
087E 0A          66              ASL
087F 8D 33 09    67              STA X2L
0882 AD 36 09    68              LDA XMIDH
0885 2A          69              ROL
0886 8D 32 09    70              STA X2H
0889             71  ;
0889 8A          72              TXA                 ;A<-YMID
088A 0A          73              ASL                 ;Y2<<-[2*YMID]
088B 8D 3B 09    74              STA Y2L
088E A9 00       75              LDA #0
0890 8D 3E 09    76              STA YCRDF           ;YCRDF<-0
0893 2A          77              ROL
0894 8D 3A 09    78              STA Y2H
0897             79  ;
0897 8A          80              TXA                 ;A<-YMID
0898 18          81              CLC                 ;XY<<-[XMID+YMID]
0899 6D 37 09    82              ADC XMIDL
089C 8D 39 09    83              STA XYL
089F A9 00       84              LDA #0
08A1 6D 36 09    85              ADC XMIDH
08A4 8D 38 09    86              STA XYH
08A7             87  ;
08A7 AD 37 09    88              LDA XMIDL           ;XCRD<<-[XMID-R2]
08AA 38          89              SEC
08AB ED 30 09    90              SBC R2
08AE 8D 35 09    91              STA XCRDL
08B1 AD 36 09    92              LDA XMIDH
08B4 E9 00       93              SBC #0
08B6 8D 34 09    94              STA XCRDH
08B9             95  ;
08B9 AD 35 09    96  NXPOINT     LDA XCRDL           ;H<<-XCRD
08BC 8D 2E 09    97              STA HL
08BF AD 34 09    98              LDA XCRDH
08C2 8D 2D 09    99              STA HH
08C5            100  ;
08C5 AD 3D 09   101              LDA YCRD            ;V<<-YCRD
08C8 8D 31 09   102              STA V
08CB            103  ;
08CB 20 2C 08   104              JSR PLOTFOUR        ;PLOT SET OF POINTS
08CE            105  ;
08CE AD 39 09   106              LDA XYL             ;H<<-[XY-YCRD]
08D1 38         107              SEC
08D2 ED 3D 09   108              SBC YCRD
```

```
08D5 8D 2E 09   109             STA HL
08D8 AD 38 09   110             LDA XYH
08DB E9 00      111             SBC #0
08DD 8D 2D 09   112             STA HH
08E0            113 ;
08E0 AD 39 09   114             LDA XYL             ;V<-[XY-XCRD]
08E3 38         115             SEC
08E4 ED 35 09   116             SBC XCRDL
08E7 8D 31 09   117             STA V
08EA            118 ;
08EA 20 2C 08   119             JSR PLOTFOUR        ;PLOT REMAINING POINTS
08ED            120 ;
08ED AD 3C 09   121             LDA YMID            ;DIVISOR<-[YMID-YCRD]
08F0 38         122             SEC
08F1 ED 3D 09   123             SBC YCRD
08F4 85 1A      124             STA DIVISOR
08F6            125 ;
08F6 AD 37 09   126             LDA XMIDL           ;DIVIDEND<-[XMID-XCRD]*256
08F9 38         127             SEC
08FA ED 35 09   128             SBC XCRDL
08FD            129 ;
08FD A2 08      130             LDX #8              ;BITCT<-8
08FF            131 ;
08FF A0 00      132             LDY #0              ;CLEAR QUOTIENT
0901 84 1B      133             STY QUOTIENT
0903            134 ;
0903 06 1B      135 DIVIDE1     ASL QUOTIENT
0905 2A         136             ROL
0906 C5 1A      137             CMP DIVISOR
0908 90 04      138             BCC DIVIDE2
090A E5 1A      139             SBC DIVISOR
090C E6 1B      140             INC QUOTIENT
090E CA         141 DIVIDE2     DEX
090F D0 F2      142             BNE DIVIDE1
0911            143 ;
0911 AD 3E 09   144             LDA YCRDF           ;YCRDF<-[YCRDF-QUOTIENT]
0914 38         145             SEC
0915 E5 1B      146             SBC QUOTIENT
0917 8D 3E 09   147             STA YCRDF
091A            148 ;
091A B0 03      149             BCS CKX             ;YCRDY< -1?
091C CE 3D 09   150             DEC YCRD            ;YES, YCRD<-[YCRD-1]
091F            151 ;
091F EE 35 09   152 CKX         INC XCRDL           ;XCRD<<-[XCRD+1]
0922 D0 03      153             BNE CKX1
0924 EE 34 09   154             INC XCRDH
0927            155 ;
0927 CE 30 09   156 CKX1        DEC R2              ;TALLY R2
092A 10 8D      157             BPL NXPOINT         ;REPEAT UNTIL XCRD=XMID
092C 60         158             RTS
092D            159 ;
092E            160 HH          DFS 1               ;HORIZONTAL PLOT VALUE
092F            161 HL          DFS 1
```

(Continued on page 28)

Circles (Listing continued, text begins on page 18)

Listing Two

```
0930              162  R         DFS 1              ;RADIUS
0931              163  R2        DFS 1              ;HOLDS R/SQR(2)
0932              164  V         DFS 1              ;VERTICAL PLOT
0933              165  X2H       DFS 1              ;HOLDS XMID*2
0934              166  X2L       DFS 1
0935              167  XCRDH     DFS 1              ;X COORDINATE
0936              168  XCRDL     DFS 1
0937              169  XMIDH     DFS 1              ;HORIZONTAL CENTER
0938              170  XMIDL     DFS 1
0939              171  XYH       DFS 1              ;HOLDS XMID+YMID
093A              172  XYL       DFS 1
093B              173  Y2H       DFS 1              ;HOLDS YMID*2
093C              174  Y2L       DFS 1
093D              175  YMID      DFS 1              ;VERTICAL CENTER
093E              176  YCRD      DFS 1              ;Y COORDINATE
093F              177  YCRDF     DFS 1              ;FRACTIONAL PART OF YCRD
093F              178  ;
001A              179  DIVISOR   EPZ $1A
C052              180  FULL      EQU $C052
C050              181  GRAPHICS  EQU $C050
00E4              182  HCOLOR    EPZ $E4
C057              183  HIRES     EQU $C057
F457              184  HPLOT     EQU $F457          ;APPLESOFT HIRES PLOT
00E6              185  HPAGE     EPZ $E6
00E1              186  LASTHH    EPZ $E1            ;HORIZ COORD OF LAST HPLOT
00E0              187  LASTHL    EPZ $E0
00E2              188  LASTV     EPZ $E2            ;VERT COORD OF LAST HPLOT
C054              189  PAGE1     EQU $C054
001B              190  QUOTIENT  EPZ $1B
093F              191  ;
093F              192            END

***** END OF ASSEMBLY
```

End Listing Two

Listing Three

```
10    '*** CIRCLE PLOT ***
20    '
30    INPUT "CENTER, RADIUS"; CX,CY,R: X=R: Y=0: A=-2*X+1:
      B=1: GOSUB 70: GOTO 30
40    '
50    'PLOT A POINT IN EACH OCTANT,
60    '
70    PSET(X+CX,Y+CY): PSET(Y+CX,X+CY): PSET(-Y+CX,X+CY):
      PSET(-X+CX,Y+CY): PSET(-X+CX,-Y+CY): PSET(-Y+CX,-X+CY):
      PSET(Y+CX,-X+CY): PSET(X+CX,-Y+CY)
80    '
90    'COMPUTE NEXT POINT.  F IS X^2+Y^2-R^2, A IS THE CHANGE
100   'IN X^2 WHEN X IS DECREMENTED BY 1, AND B IS THE CHANGE
110   'IN Y^2 WHEN Y IS INCREMENTED BY 1.  F IS NOT ALLOWED TO
120   'EXCEED  R; EQUIVALENTLY, THE POINT (X,Y) IS KEPT WITHIN
130   'A DISTANCE R+1/2 OF THE CIRCLE CENTER.  THE ALGORITHM
140   'IS DONE WHEN THE CHANGE IN Y^2 REACHES THE NEGATIVE OF
150   'THE CHANGE IN X^2 ( B>=-A ).
160   '
170   IF B>=-A THEN RETURN ELSE Y=Y+1: F=F+B: IF F>R THEN
      F=F+A: A=A+2: X=X+1
180   B=B+2: GOTO 70
```

End Listing Three

Circles

Listing Four

```
10 DEFINT A-Z
20 PRINT CHR$(26) 'CLEAR DUMB TTY SCREEN
50 FOR I=1 TO 11
55 AE=I*2 'WIDTH OF ELLIPSE
56 BE=I*1 'HEIGHT OF ELLIPSE
57 XC=I*4+1 'CENTER.X OF ELLIPSE
58 YC=I*1 'CENTER.Y OF ELLIPSE
60 GOSUB 1060 'PLOT A CIRCLE
70 NEXT I 'PLOT 11 CIRCLES
998 END
1050 '****** CIRCLE SUBROUTINE
1060 XF=0                    'INIT X-OFFSET
1070 YF=BE                   'INIT Y-OFFSET
1080 XD=BE*BE                'INIT COMPUTATION OF X-SQUARED
1090 YD=(2*BE-1)*AE*AE       'INIT COMPUTATION OF Y-SQUARED
1100 DX=2*BE*BE              'DEFINE DELTA-(X-SQUARED)
1110 DY=2*AE*AE              'DEFINE DELTA-(Y-SQUARED)
1120 ER=0                    'INIT ERROR (I.E. ER=AE^2*BE^2-XF^2*BE^2-YF^2*AE^2)
1130 GOSUB 1260              'PLOT THE FOUR POINTS
1140 TX=ER+XD
```

```
        : TY=ER-YD
        : TB=ER+XD-YD
1150 IF ABS(TX)>=ABS(TY) OR ABS(TX)>=ABS(TB) THEN 1170
1160 XF=XF+1
        : ER=TX
        : XD=XD+DX
        : GOTO 1220
1170 IF ABS(TY)>=ABS(TX) OR ABS(TY)>=ABS(TB) THEN 1190
1180 YF=YF-1
        : ER=TY
        : YD=YD-DY
        : GOTO 1220
1190 IF ABS(TB)>=ABS(TX) OR ABS(TB)>=ABS(TY) THEN 1210
1200 XF=XF+1
        : YF=YF-1
        : ER=TB
        : YD=YD-DY
        : XD=XD+DX
        : GOTO 1220
1210 PRINT"OOPS"; 'IF HERE THEN THERE IS A BUG.
1220 GOSUB 1260 'PLOT THE POINTS
1230 IF  YF<>0 THEN 1140
1240 RETURN
1250 '****** ROUTINE TO PLOT FOUR POINTS AT ONCE
1260 XP=XC+XF
        : YP=YC+YF
        : GOSUB 1320
1270 XP=XC+XF
        : YP=YC-YF
        : GOSUB 1320
1280 XP=XC-XF
        : YP=YC+YF
        : GOSUB 1320
1290 XP=XC-XF
        : YP=YC-YF
        : GOSUB 1320
1300 RETURN
1310 '****** ROUTINE TO PLOT A POINT ON A DUMB TERMINAL
1320 C1=YP+32
        : C2=XP+32
1330 IF YP<0 OR YP>23 OR XP<0 OR XP>79 THEN 1360
1340 PRINT CHR$(27);CHR$(61);CHR$(C1);CHR$(C2);"*";
1350 RETURN
1360 PRINT "POINT OUT OF BOUNDS"
        : STOP
```

End Listing Four

Cursor Control for Dumb Terminals

A friend of mine asked me one day, "Why should I purchase a new CRT terminal when I can buy a surplus unit for less than half the price?"

"Because," I answered, "that so-called bargain of yours is what we used to call a 'dumb' terminal. Unlike today's marvels, it has only the most basic of cursor control functions: home, line feed, carriage return, backspace, and skip (the skip function moves the cursor across the screen one column at a time without deleting any currently at that position). It is incapable of direct cursor addressing, which is required by most of the software available on the market today. In short, it is not a bargain; it is an antique."

"What," my friend asked, "should I do with this antique I just bought?"

The solution to his problem turned out to be surprisingly simple and should be of interest to anyone owning a terminal without direct cursor addressing and a microcomputer system running CP/M. (It should also interest those wanting to connect more esoteric peripherals than terminals or printers to their systems, if only to illustrate the level of complexity that can be supported.) Only the software need be modified. Even then the CP/M operating system remains absolutely standard and is still capable (assuming the IOBYTE feature has been implemented) of supporting a second terminal with direct cursor addressing without further modifications. What more could you ask for?

A terminal with direct cursor addressing traps a predefined string of ASCII characters sent from the computer to the terminal and uses the information contained in this string to position the cursor on the screen. (The string itself, of course, is not displayed.) As an example, the Televideo 950 terminal responds to the ASCII string "ESC=yx," where y is an ASCII character from "SP" (32 decimal) to "7" (55 decimal) that positions the cursor vertically and x is an ASCII character from "SP" to "o" (111 decimal) that positions the cursor horizontally. To emulate this feature, a "dumb" terminal should be able to respond to a similar string of ASCII characters and position its cursor accordingly.

While it would be nice to have the terminal perform this function through its own hardware, this is obviously impractical for a large number of users; redesigning and rewiring old printed circuit boards without schematics is a task few care to tackle! What remains is to modify the software, specifically CP/M's BIOS (Basic Input/Output System), which provides the interface between the terminal and the host computer. The CP/M operating system can then monitor what the running program is sending through the BIOS to the terminal.

When an ASCII "ESC" character is sent to the modified BIOS (from, say, a full-screen word processing program), it will store this character temporarily rather than output it through the serial port to the terminal. When the next character sent by the program is received, the BIOS will check to see if it is an ASCII "=" character. If it isn't, the BIOS will output both the stored "ESC" character and the received character to the terminal, then continue to output characters received from the program until the next "ESC" character arrives; otherwise, it will recognize the two characters as the start of a direct cursor positioning command, discard them, and wait for the next character.

Once this character is received, the BIOS will decode it, output a single ASCII control character to home the cursor on the terminal, and output the required number of line feeds to position the cursor vertically. Then it will trap the next character sent by the program, decode it, and output the required number of skip control characters to position the cursor horizontally. Once the cursor is positioned where required, everything returns to normal: all ASCII characters sent to the BIOS by the program will be sent to the terminal after being monitored for an "ESC" character and the possible start of another direct cursor positioning command.

It sounds complicated, but it is simple to program and works beautifully as long as you operate the terminal at 9600 baud. At this speed the cursor will take, in the worst case, approximately one-tenth of a second to traverse the screen (from the home position to the 24th row and 80th column). The motion of the cursor is perceptible but not objectionable; a slower baud rate, however, is an invitation to crossed eyeballs from watching the cursor bounce around the screen!

The entire software routine can be written as a peripheral driver and added to your BIOS in exactly the same way as you would add a printer or modem to your system.

If your version of CP/M doesn't include the IOBYTE feature, then the accompanying code (Listing One, page 33) should be substituted for your existing terminal driver (usually the physical device TTY: or CRT: in the BIOS source code). Once this is done, your version of CP/M will be customized for the "dumb" terminal being used as its console device. When a newer model terminal is purchased, the BIOS must be customized once again to support it as the console device instead of the old terminal.

Fortunately, most commercial versions of CP/M do include the IOBYTE feature. In this case, the accompanying code should be placed in whatever physical I/O device driver routine isn't being used (usually the user-defined console device UC1:). By leaving the TTY: or CRT: physical device code (whichever currently supports the computer's terminal device) intact or by modifying it to support the terminal you eventually intend to purchase, you can switch between a "dumb" terminal and a modern direct-cursor-addressable terminal simply by using the STAT CON:=UC1: and STAT CON:=TTY: (or STAT CON:=CRT:, as applicable) commands. You can also modify your BIOS to initially set the IOBYTE so that CON:=UC1: after each cold boot.

Because the procedure for modifying CP/M's BIOS may vary, depending on the idiosyncracies of your particular system, I will not describe it in this article. If you haven't interfaced a peripheral to your system before, I suggest that you have your vendor or an experienced friend perform the modifications indicated in Listing One; otherwise, you should consult some of the excellent books on CP/M that are available (the finest I know of is David Cortesi's *Inside CP/M – A Guide for Users and Programmers,* CBS College Publishing, 1982).

By the way, some excellent bargains are available on the surplus computer equipment market – if you know exactly what you are buying. Otherwise, caveat emptor, or "let the buyer beware!" DDJ

by Ian Ashdown

Ian Ashdown, byHeart Software, 2-2016 West First Avenue, Vancouver, B.C., V6J 1G8, Canada.

Cursor Control Listing (Text begins on page 32)

```
;*****************************************************************
;*                                                               *
;*                    CP/M BIOS Modifications                    *
;*                                                               *
;*     Author: Ian Ashdown                                       *
;*             byHeart Software                                  *
;*             2 - 2016 West First Avenue                        *
;*             Vancouver, B.C. V6J 1G8                           *
;*                                                               *
;*     Date:   February 10th, 1983                               *
;*                                                               *
;*****************************************************************
;
; The following code has been added to allow the ANAND II "dumb"
; terminal to emulate a direct-addressable cursor terminal
; (specifically the TELEVIDEO 950). It does this by trapping an
; "ESC=yx" control code from a program and interpreting "y" as
; the vertical row address (where 20H is row #1) and "x" as the
; horizontal column address of the cursor (where 20H is column
; #1). The BIOS then homes the cursor and outputs as many line
; feeds and skips as is required to position the cursor without
; changing the current screen display. At 9600 baud the worst
; case transit time of the cursor (from "home" to the 24th row
; and 80th column) is approximately 1/10 second.
;
;*************************** EQUATES *****************************
;
; NOTE: The following equates are independent of the system used.
;       However, they may already be defined in your BIOS source
;       code.
;
LF      EQU     0AH     ;ASCII line feed
CR      EQU     0DH     ;ASCII carriage return
ESC     EQU     1BH     ;ASCII escape
;
; NOTE: The following equates are specific to my California
;       Computer System's serial I/O port and the control codes
;       recognized by the ANAND II terminal. Replace them
;       with those required by your system.
;
RESET   EQU     19H     ;ANAND II "home cursor" control char.
SKIP    EQU     14H     ;ANAND II "cursor forward" control char.
SDATA   EQU     20H     ;SERIAL DATA PORT
SLSTAT  EQU     SDATA+5 ;Serial line status port
RXRDY   EQU     00000001B  ;Receive data available bit
TXMTY   EQU     00100000B  ;Transmit buffer empty bit
;
;********** NEW PHYSICAL DEVICE UC1: DRIVER ROUTINE ************
;
; I/O DRIVERS FOR THE 8250 ASYNC COMM ELEMENT
; (PHYSICAL DEVICE UC1:)
;
```

```
;  NOTE: The following CUST1: and CUSI1: subroutines are vendor-
;        dependent code (see COMMENTARY at the end of this
;        listing), and should be replaced with the TTY: or CRT:
;        driver code specific to your system.
;
CUST1:  IN      SLSTAT  ;Get 8250 line status
        ANI     RXRDY   ;See if receive data available
        RZ              ;Return if not
        ADI     OFFH AND NOT RXRDY      ;Flag that data is
        RET                             ;available
;
CUSI1:  CALL    CUST1   ;Get 8250 line status
        JZ      CUSI1   ;Loop until data is in
        IN      SDATA   ;Read the data
        RET
;
;  NOTE: The following subroutines CUSO1: through HORZOUT:
;        inclusive are independent of the system used.
;
CUSO1:  LDA     ESCPTR  ;Get escape sequence pointer
        CPI     0       ;Jump to TESTCH if this isn't an
        JZ      TESTCH  ;escape sequence
        CPI     1       ;Jump to TESTEQU if this is an escape
        JZ      TESTEQU ;sequence
        CPI     2       ;Jump to VERTIN if true (character is
        JZ      VERTIN  ;row address of cursor)
        JMP     HORZIN  ;Jump to HORZIN (character is column
                        ;address of cursor)
TESTCH: MOV     A,C     ;Get character from register C
        CPI     ESC     ;Jump to NOTESC if it isn't an ASCII ESC
        JNZ     NOTESC
        MVI     A,1     ;Set escape sequence pointer to 1 to
        STA     ESCPTR  ;indicate ASCII ESC character
        RET
NOTESC: CALL    CUSOUT  ;Output character
        RET
;
TESTEQU:MOV     A,C     ;Get character from register C
        CPI     '='     ;Jump to NOTEQU if it isn't an ASCII "="
        JNZ     NOTEQU
        MVI     A,2     ;Set escape sequence pointer to 2 to
        STA     ESCPTR  ;indicate ASCII "=" character
        RET
NOTEQU: MVI     A,0     ;Set escape sequence pointer to 0 (false
        STA     ESCPTR  ;alarm)
        MVI     A,ESC   ;Output previous ASCII ESC character
        CALL    CUSOUT
        MOV     A,C     ;Get current character from register C
        CALL    CUSOUT  ;Output it
        RET
;
```

(Continued on page 36)

Cursor Control Listing (Listing continued, text begins on page 32)

```
VERTIN: MOV     A,C       ;Get character from register C
        SUI     1FH       ;Subtract cursor address bias (20H is row
                          ;number 1) to get row address counter
        MOV     C,A       ;Store row address counter in register C
        MVI     A,3       ;Set escape sequence pointer to 3
        STA     ESCPTR
        MVI     A,RESET   ;Output "reset cursor" control character
        CALL    CUSOUT
VERTOUT:DCR     C         ;Decrement row address
        RZ                ;Return if done
        MVI     A,LF      ;Output a line feed
        CALL    CUSOUT
        JMP     VERTOUT   ;Loop until done
;
HORZIN: MOV     A,C       ;Get character from register C
        SUI     1FH       ;Subtract cursor address bias (20H is
        MOV     C,A       ;column number 1) to get column counter
        XRA     A         ;Set escape sequence pointer to 0
        STA     ESCPTR
HORZOUT:DCR     C         ;Decrement column counter
        RZ                ;Return if done
        MVI     A,SKIP    ;Output a "cursor skip" control character
        CALL    CUSOUT
        JMP     HORZOUT   ;Loop until done
;
ESCPTR  DB      0         ;Escape sequence pointer
;
; NOTE: The following CUSTOUT: subroutine is vendor-dependent
;       code (see COMMENTARY at the end of this listing), and
;       should be replaced with the TTY: or CRT: driver code
;       specific to your system.
;
CUSOUT: PUSH    PSW       ;Save character
CUSOUT1:CALL    CUSOST    ;Get 8250 line status
        JZ      CUSOUT1   ;Wait until one of the registers empties
        POP     PSW       ;Recall character
        OUT     SDATA     ;Output the data
        RET
;
CUSOST: IN      SLSTAT    ;Get 8250 line status
        ANI     TXMTY     ;Isolate TX buffer empty bit
        RZ                ;Return if not empty
        ADI     OFFH AND NOT TXMTY      ;Flag the empty state
        RET
;
; ************************ DELETED CODE ****************************
;
```

```
; The following code has been deleted:
;
;CUSI1: EQU     IOER    ;UNASSIGNED USER CONSOLE (INPUT)
;CUSO1: EQU     IOER    ;UNASSIGNED USER CONSOLE (OUTPUT)
;CUST1: EQU     IOER
;
; ************************ COMMENTARY ****************************
;
; The following California Computer Systems vendor code for a
; typical serial interface to a terminal has been added for
; information only as an example to show where the I/O portion
; of the above UC1: code was derived from. Most CP/M TTY: and
; CRT: physical device drivers will be similar to this code.
;
; I/O DRIVERS FOR THE 8250 ASYNC COMM ELEMENT
; (PHYSICAL DEVICE TTY:)
;
;TTST:  IN      SLSTAT  ;GET 8250 LINE STATUS
;       ANI     RXRDY   ;SEE IF RECEIVE DATA AVAILABLE
;       RZ              ;RETURN IF NOT
;       ADI     OFFH AND NOT RXRDY      ;FLAG THAT DATA IS
;       RET                             ;AVAILABLE
;
;TTYIN: CALL    TTST    ;GET 8250 LINE STATUS
;       JZ      TTYIN   ;LOOP UNTIL DATA IS IN
;       IN      SDATA   ;READ THE DATA
;       RET
;
;TTOST: IN      SLSTAT  ;GET 8250 LINE STATUS
;       ANI     TXMTY   ;ISOLATE TX BUFFER EMPTY BIT
;       RZ              ;RETURN IF NOT EMPTY
;       ADI     OFFH AND NOT TXMTY      ;FLAG THE EMPTY STATE
;       RET
;
;TTYOUT:CALL    TTOST   ;GET 8250 LINE STATUS
;       JZ      TTYOUT  ;WAIT UNTIL ONE OF THE REGISTERS EMPTIES
;       MOV     A,C     ;MOVE THE DATA OVER
;       OUT     SDATA   ;OUTPUT THE DATA
;       RET
;
; ************************ END OF LISTING ************************
```

End Listing

Dysan's Digital Diagnostic Diskette

Dilemma: You've been seeing more flexible disk errors lately, especially when swapping disks between your drives. But they seem to work properly most of the time, so you struggle on, tolerating the occasional loss of your data, frustration, and wasted time. After all, you're not sure anything is wrong with the drives; to find out, you'd have to remove one from the system and ship it off to a stranger who probably makes money five times as fast as you do. If the problem persisted after the drive came back, would you send another one off to be checked?

Deliverance: Now, for less than what you'd probably spend having one drive professionally checked, you can equip yourself to verify the operation of all your flexible disk drives, anytime you suspect a problem, without disassembling or altering your system! But you'll have to begin now, before you have serious operating problems – partly because you'll have to customize a public domain program to fit the quirks of your terminal and disk controller, but mainly because you'll want to have a "before" picture of your drives' operation to help you recognize any real problems that may appear later.

You thought a drive exerciser and dual-channel triggered sweep oscilloscope were essential for flexible drive alignment? Well, the Dysan Corporation, which makes the Analog Alignment Diskette™ for scope alignment, now offers a Digital Diagnostic Diskette™. Using your own computer system and the program described in this article, you can verify your drives' alignment to within one-half milli-inch of the expensive analog method. Plus, you can check the spindle bearings and clamping mechanism, verify spindle speed, check positioner hysteresis (worn lead screw, bad stepper motor, etc.), verify read/write head azimuth, and check index-hold timing, all under your own operating conditions.

by Loren Amelang

Loren Amelang, Box 24, Philo, CA 95466-0024.

"Analog Alignment Diskette," "AAD," "Digital Diagnostic Diskette," and "DDD" are trademarks, and "Dysan" is a registered trademark, of Dysan Corporation.

What the DDD Is

The Digital Diagnostic Diskette (DDD™) looks like any other floppy disk. It has ordinary formatting, and conventional data are recorded on it. Figure 1 (page 42) shows a small sample of disk data patterns to scale. You could probably run your disk copy program and duplicate it. But your copy would not be another DDD! The digital alignment concept depends upon having the data fields of the DDD carefully displaced from their proper location or azimuth by known amounts.

For example, let's look at the centering test. Three "alternate offset" tracks on the DDD are offset 7, 8, and 9 milli-inches from the official center of the track (see Figure 2, page 42). (All values given are for 48 track-per-inch drives; if you have 96 or 100 tpi drives, tolerances are tighter and thus offsets are smaller.) All odd-numbered sectors are displaced toward the center of the disk, and even-numbered sectors toward the outside edge. Your head gap is designed to pick up a 12 milli-inch swath of data, so if it is perfectly aligned but the data are offset 7 milli-inches, it "sees" only 5 milli-inches of the data. In analog terms, the amplitude of the signal is reduced drastically; in digital terms, either your drive's electronics can make sense of the weak signal, or you get a read error. As you move to the 9 milli-inch offset track, the width of the recorded data still under a properly positioned head is reduced to 3 milli-inches; if your spindle bearings have .003 inches of play, the signal may disappear entirely at certain points. If your head is dirty, and the accumulated oxide holds it away from proper contact with the media, the already reduced signal amplitude will be cut still further. On the other hand, if your system can reliably read data that are offset 9 milli-inches in either direction, you can be pretty sure that your drive is mechanically sound, its head is reasonably aligned, and its electronics are not being swamped by stray electrical or magnetic fields.

If you want to pin down more exactly the radial alignment of your read/write head, the DDD has "progressive offset" tracks (Figure 3, page 44). Here sector one is offset 1 milli-inch toward the spindle, sector two 1 milli-inch away, sector three 2 milli-inches toward the spindle, and so on out to 13 milli-inches in either direction. (This applies to Revision B; Revision A DDD had ±10 milli-inch maximum offsets.) You read through the odd-numbered and then the even-numbered sectors, noting the first sectors that cannot be read. This information can be translated into a graphic display of your effective head width and its position relative to proper alignment. Electrical or magnetic noise in your environment, "dirt" on your head, or improper head loading will narrow the effective head width. A major advantage of the DDD is that you can run this test on six different tracks spaced across the diskette; analog alignment uses only one central track, with the hope the rest will follow.

The "azimuth rotation" track is another progressive track used to check whether your read/write head gap is adversely skewed relative to the data track. Here data are recorded on the radial centerline of the track, but the head azimuth is progressively rotated away from normal (Figure 4, page 46). If read errors occur symmetrically, your effective head azimuth is correct. If your system fails in one direction before the other, you may improve performance by altering your head azimuth toward the earlier angle. Again, a decrease in azimuth range could indicate a dirty head or environmental noise.

Other recorded DDD tracks contain a special formatting pattern for timing the passage of the index hole past your sensor and for detecting how long it takes your head to settle into position after it is loaded. Finally, some tracks are marked "user area," where you can store the program you'll need to read the DDD and display the results. *Don't* try to do this under CP/M! You'll overwrite other parts of the DDD and destroy it! If you are going to make use of the "user area," you'll need a "disk doctor" program that can write to specific tracks of the disk, and you'll need some way to load and run what you've stored without involving CP/M.

How does Dysan Corporation manage to create a disk with precisely misaligned information on it? To begin, they take their *premium diskettes* and screen them: any alignment diskette must meet high standards for dimensional stability and uniformity of the recording surface. The DDD must be error-free, not only in the normal tracks but "in the cracks" as well. Critical information is going to be recorded completely off track for some of the tests.

Several million dollars went into the computer-controlled, laser-interferometer-calibrated trackwriters and the environ-

mentally standardized room that houses them. Once the blank diskettes have acclimated to the standard environment, they are individually written; the track-writers actually move the write heads to all those different off-track positions. These same machines write the Analog Alignment Diskettes used by manufacturers and calibration services. The same precise standards are met, both for the media and for the recorded information. The only advantage of analog alignment is that it is not limited to resolving the 1 milli-inch steps of the DDD but can get closer to the exact center of track. Of course, the DDD has the advantage of being able to check alignment on six different tracks and may reveal more than one-half milli-inch of variation between the inner and outer track centers.

Using the DDD

Dysan Corporation has been making the DDD for about two years. It was conceived as a way for manufacturers to verify the operation of completed systems quickly, without having to reach any interior test points. Field service personnel have been using it to diagnose drive problems at the customer's location. Recently, the company has been promoting the DDD to end users at various computer shows.

To assist the end user in creating a program that reads the DDD and displays the results of the various tests, the Customer Engineering Division of Dysan Corporation developed the original (version 1.0) DDD program. Designed to run on their "Single Card Computer" with a Hazeltine 1500 terminal, version 1.0 employs direct-memory-access (dma) transfers of read data and requires an 8080 macro assembler for assembly. Luckily for those of us with other types of equipment, constants are clearly defined at the beginning of the code, and the hardware-dependent routines are collected at the end of the code. I found the macro facility was rarely used, so I expanded the few examples into code that ASM can handle. Since my terminal does not have a clear-to-end-of-line function, I inserted a conditional that either prints a string of blanks when it is absolutely essential or allows those who have the function to benefit from its greater speed.

This still leaves a few basic requirements: The program depends heavily on being able to clear your terminal screen and address your cursor. If you have a really dumb terminal, you will have to completely rethink the graphic display parts of the code. The program, as written, commands a Western Digital 1793 floppy disk controller chip; if your system does not use this chip, you will have to understand both the 1793 and your own controller well enough to translate commands between them. Certain controller chips are not capable of performing all the commands the program uses, so some ingenuity may be required. Different systems have different ways of selecting which drive is to be tested. I have added to version 1.0 a routine to select any of four drives connected to my CCS 2422 controller. You'll have to understand how your system goes about selecting drives. Finally, as should be obvious, you'll need to be able to deal with 8080 assembly language in order to get the DDD program working.

Nothing inherent in the DDD limits it to CP/M or 8080 systems. It is available in models for 8-inch or 5¼-inch drives; single or double sided; single or double density; 48, 96, or 100 spi. Since the DDD program has been placed in the public domain, it may be used as an example to construct versions for other machines, processors, and operating systems. If you develop a version for a popular computer, I hope you'll make it available on a bulletin board or software exchange.

If you'd like to take the mystery out of your flexible disk system, and you're willing to be guided through some hardware-intensive assembly language, here's your agenda. First, order a DDD. If you have double-sided drives, you'll need the double-sided DDD in order to test both heads. If all your drives have double-density capability, you'll want to test them in this more stringent mode. If some are single density only and your doubles can read single, perhaps you'd prefer a single DDD for all of them. You'll need to specify the disk size and number of tracks per inch. Call your local Dysan representative, or call the factory at (408) 988-3472 or (800) 551-9000 outside Northern California, and be prepared to spend $30.00 for a single-sided DDD or $40.00 for a double-sided DDD, plus shipping and applicable taxes. For more information on DDD applications, call Dysan's Tech-Line at (408) 734-1624.

Next, you'll need machine-readable source code for the program that reads the DDD and displays the results. DDD11.ASM (or DDD11.AQM in the "squeezed" version) may be available on your local RCPM system. If you have a phone line, modem, XMODEM-Protocol transfer utility, and "Unsqueeze" if DDD11.AQM is used there, you can copy the program yourself for free. If you must call long distance, be warned that DDD11.ASM is a 47K file. You might be better off downloading it from Micronet's CP/M Interest Group "Access area 1." If you prefer to purchase the code on disk, contact your favorite source for CP/MUG disks or write CP/MUG, 1651 Third Avenue, New York, NY 10028. Should all CA 95466-0024, and enclose $10.00. Listing One (page 48) contains a slightly abbreviated version of DDD11.ASM, called DDDCompact.ASM.

Finally, you'll need some information about your hardware. Check your terminal's manual to see if it gives clear-screen and addressable cursor-control commands. See if your disk controller manual gives control port addresses, command formats, status codes, and flow charts of command execution. If the information isn't there, ask the manufacturer if more detailed programming information is available. While a source listing of your CP/M BIOS would probably reveal what you'll need to know about the disk controller, this will be more difficult to decipher. If your hardware is based on an LSI controller chip, the chip manufacturer's data sheets will be very helpful. Save yourself frustration by collecting your hardware information before you begin – while you wait for the DDD disk and DDD11.ASM to arrive.

Customizing DDD11.ASM for Your System

Once you've chosen which DDD format you need, collected your hardware manuals, and are seated at your keyboard with DDD11.ASM in your editor, we can begin. You've read the headers down to "Program Source Code." The first instruction to the assembler is "wboot equ 0" – this is nearly always true. If you happen to be dealing with a nonstandard CP/M system, you probably are well aware of the fact; you've needed special versions of programs from day one with your system and will have to modify my recommendations as we proceed.

Next come the "Conditionals." Here you must decide how committed you are to good programming style. The quick way to customize a program is to step through the code and change only a few magic numbers until it works. The right way, which will pay off handsomely if you should want to give the program to a friend or need to understand what you did come next year, is to create conditional assembly options here for both your terminal and your disk controller. Whenever you need to alter a bit of code in the body of the program, insert "if My____," enter your changes, and then write "endif." By setting "My____" equ "true" in the "Conditionals" section, you insert all your changes into the assembled code. If your friend wants the original code back, he or she can remove all your changes by setting "My____" equ "false." Unless you happen to have one of the existing conditional systems, set these conditionals equ "false," add similar lines for your terminal and controller, and set yours equ "true." Set "lineclr" true or false according to whether your terminal can clear-to-end-

of-line with a single command.

The "Macro Call" section has been commented out, and all examples of its use have been expanded by hand, so that you can use a simple assembler like ASM. Ignore it. Under "Console Functions" comes your first chance to say "if My-terminal." Set as many of the variables as your terminal uses to the values given in your manual. Don't forget the "endif." Check the ASCII code equates, especially backspace, for compatibility with your terminal.

In "Floppy Controller Port Equates," you tell the program where to send commands for the disk controller. Set up a conditional for your controller; you can probably copy these addresses directly from your BIOS source, if you have it. Note how the base address is given in absolute numbers, and all the other addresses are given as offsets from the base. Most systems with a 1793 controller chip will have these same offsets from whatever base address the system uses.

In the "if Dysan" conditional, "dma" is a port address where commands to the

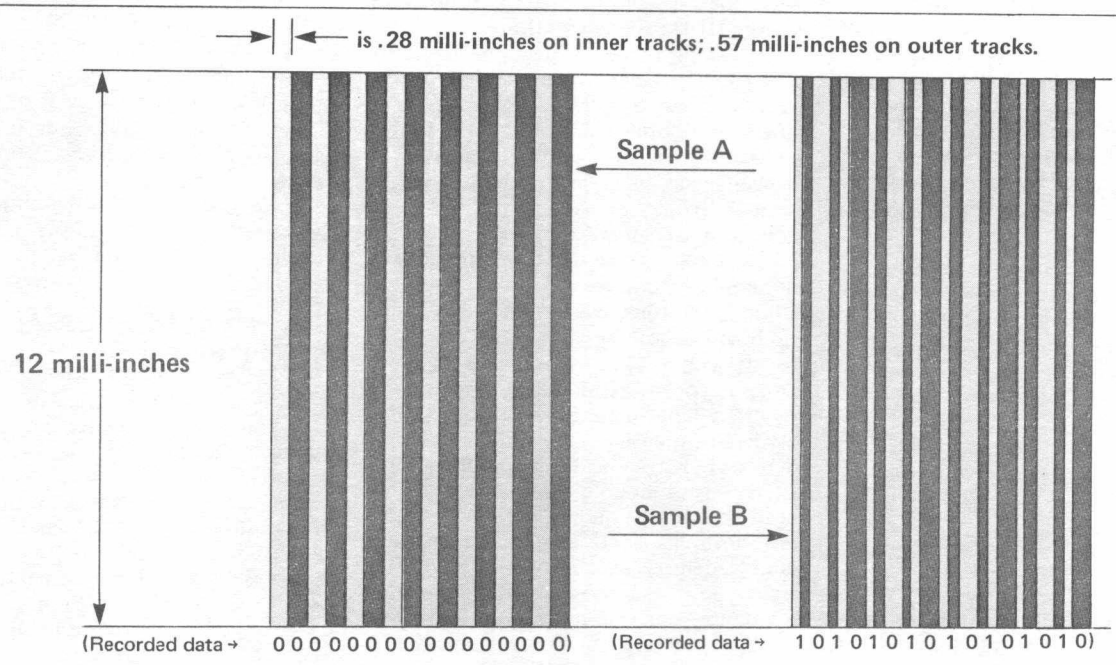


Figure 1.

The actual magnetic pattern on an 8-inch diskette, to scale.

The black lines are *not* on the diskette. There are no prearranged "boxes" to be filled in, only uniform magnetic particles. When you write on the blank diskette, you arrange these particles in alternating stripes of "north" and "south" poles (the light and dark areas in the figure). It is not the polarity itself but the *change* in polarity that is significant. Sample A is a two-byte string of "0" bits, which means that there are no (zero) transitions with a bit space. Sample B alternates between "1" and "0" bits; a transition within a bit space is read as a "1."

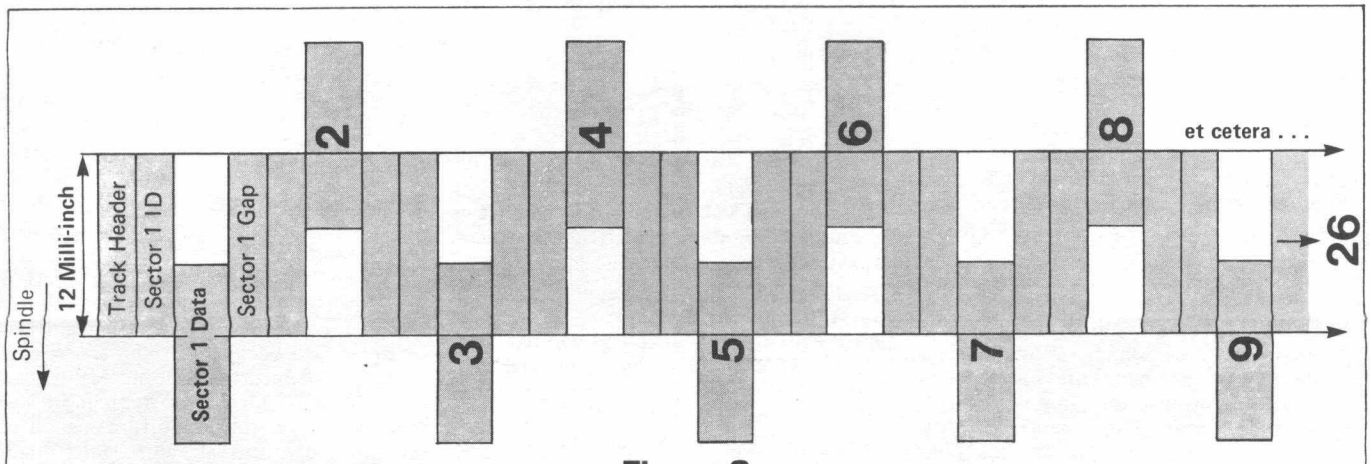


Figure 2.

An alternative offset track.

Each data sector is 128 bytes, while sector IDs are 31 bytes and sector gaps are 29 bytes or less. The track header is 73 bytes. The circular track is shown here stretched out straight for convenience — tracks form complete circles on diskettes.

direct memory access controller are sent. Each time this controller is asked to read the disk, it is sent an address in memory at which to begin storing what is read. It reads the disk and puts the data directly into memory, while the CPU does other tasks, by using the slivers of time between CPU memory accesses. In my CCS conditional, "dma" is a memory address where a pointer to a memory buffer is stored. In this non-dma or programmed I/O transfer system, after telling the controller to read the disk, the CPU must be ready to grab each byte as it is read and to move it into memory before the next byte is ready. The CPU looks at "dma" to find out where in memory to begin storing the read data. Obviously, the CPU cannot do anything else while it is busy moving in data; in fact, to keep up with a double-density 8-inch disk it must run at least at 4 MHz.

"Floppy Controller Commands" is another section you might copy from your BIOS source. Command "sdma" tells the Dysan controller to begin transferring data; if you have a dma-type controller, it should have a similar command. Variable "fdelay" is used in the "delay" routine, where time is being carefully counted. Instead of issuing a command to the controller and checking to see if it is processed, this routine dispenses the delay time in known chunks of 50 microseconds. If you don't have a 4 MHz Z80, and you want the timing routines to work, "fdelay" and the "delay" routine will have to be juggled until the loop takes 50 microseconds. "Indxbyt," "busybyt," and "drqbyt" are masks that select the appropriate bit from the status register of the controller chip. For example, if the status register is read, and the logical "and" function is done with "busybyt," a 0 will result unless the bit signifying "busy" is set high. A bit-by-bit explanation of status register contents should be available in your controller chip data sheet. The other variables in this section go directly to the controller chip and can be determined from the manufacturer's data sheet, if not from your *BIOS source*.

In "DDD Diskette Variables," the total number of tracks and the number of sectors per track in your chosen format are set. Note that several other parameters are defined in terms of sectors and need not be changed. "Ref" is a magic number that will be discussed later under "Index Timing." Ignore it for now. Those of you with nonstandard CP/M systems will have to change the "org" address. Otherwise we're already down to the actual program!

Step on down to "Clear Console Display" – the code in between should run unchanged on any CP/M system. If your terminal, like the Hazeltine, needs an extra lead-in byte ahead of the actual clear-screen command, you'll need to put in an "if Myterminal" here. "Clear Display Line" has two conditionals: one for terminals with a clear-to-end-of-line command, which is active if "lineclr equ true," and another that prints a string of

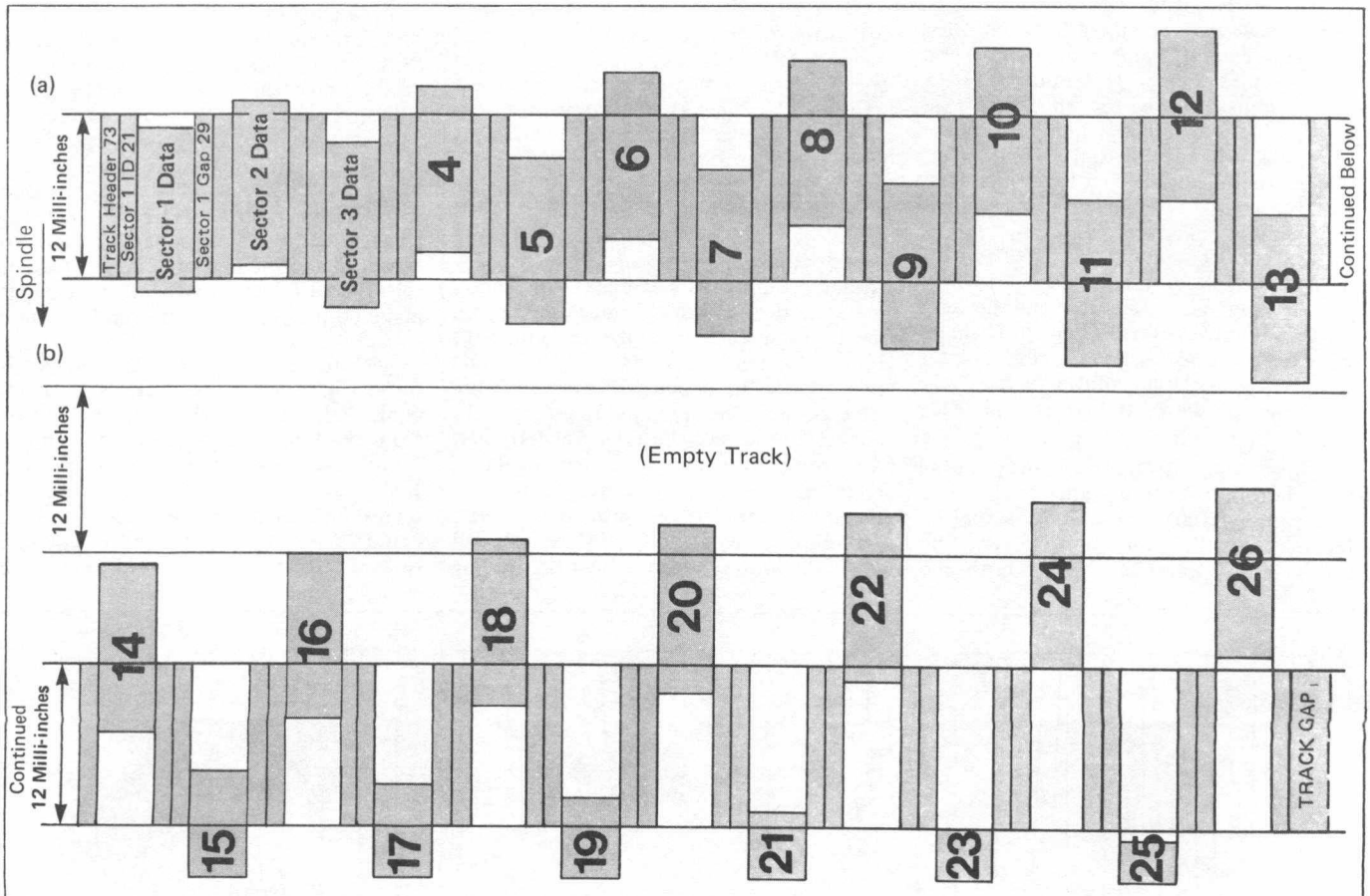


Figure 3.

A progressive offset track.

Vertical scale is 75 to 150 times horizontal, depending on the track number. The entire track is displayed here, split into two parts: sectors 14 through 26 (at (b), the bottom part of the figure) are displayed with a neighboring empty track in order to show how the offset sectors begin to bleed out into the adjacent empty tracks; sectors 1 through 13, at (a), are shown without neighboring tracks.

blanks for terminals without this feature. If you've set "lineclr" true, and if your terminal needs a lead-in byte or other special treatment, insert an "if Myterminal AND lineclr" as I have for the Hazeltine. Again, in "Print String of Text," insert a conditional under "text1" for lead-in type terminals. If your terminal should need the row position sent before the column position, accomplish that here by incrementing the HL register pair twice after the position cursor command (to get the row position) and then decrementing HL to get back to the column position. This code section assumes that the upper left-hand corner of your screen is called 0,0. If your terminal places 0,0 somewhere else, you'll have to add a constant bias to the coordinates before you send them.

"Index Timing" may require major changes. In the Dysan dma system, the command "read address" is sent to the dma controller, "start dma" is sent, and the CPU is set to counting the time until the controller is finished doing its direct memory access. If you have a dma-type controller, you'll use this scheme as it is. If you have a programmed I/O controller, the CPU cannot transfer data and count time simultaneously. My solution, in the CCS conditional, is to send the "read address" command and immediately set the CPU to counting time until "drq" comes true. When the first byte of address data is ready to be transferred to memory, the disk controller sets "drq" true, telling the CPU to "come and get it." The CPU immediately determines (lhld dma) where to put the byte, reads it from the data port, and moves it to memory. It increments the memory address and goes after the next byte. "Read address" reads only six bytes of data, so I wrote six reads inline.

Since the non-dma CPU only counts time before the data transfer, the time counting loop after that code is for the Dysan controller only. But because the Dysan dma controller counts both the time from the "read address" command to the first data byte and the time needed to transfer six bytes of data, it will get a longer time count. This is where "ref" comes in. "Ref" is a magic number or, more bluntly, a fudge factor, which makes the final count read however we want it. Since "ref" is subtracted from the total time count, the non-dma controller will require a smaller "ref" value than the dma controller. Dysan's value is experimentally set with their Analog Alignment Diskette, whose track has a standardized 200 microsecond time from index hole to address mark. My value of 24EH is estimated to make my DDD also give about 200 microseconds on my drives – hardly a precise method, but a relative measure is better than no measure at all.

"Index Timing" is the first section to introduce the problem of time counting. You've undoubtedly noticed that many of the instructions are followed by numbers in the comment field. These numbers give the execution times in microseconds for those instructions on a 4 MHz Z80. If your CPU is clocked at 2 MHz, the instructions will take twice as long. But disk reads of data happen at the speed of the moving diskette, regardless of the CPU clock. If you intend to properly adapt this routine to a different clock rate, you will have to manipulate not only the timings here, but also the delay provided by "delay" under "FDC Delay to Process Commands" and the loop count "fdelay" in "Floppy Controller Commands." Unless you enjoy mathematical puzzles, I suggest you join me in a cut-and-try fudge. In "Index to Index Timing" and its "RPM Timing Loop," no disk reads are involved, only time counting between index pulses. Here a 2 MHz processor should get exactly half the proper time, and a mathematical correction is reasonably simple. Translate the instruction times into their corresponding values at your clock speed, and set fewer loop passes or adjust the trailing "inx d" instructions to give the required 100 microsecond loop.

"Drive Select" is your chance to be original. The version 1.0 option simply selects one drive only. My CCS option can select one of four drives connected to a CCS 2422 controller, but it does not deal with double-sided drives. The code down to "jmp select" should be pretty universal. After that, you'll have to determine from your manual (or crib from your BIOS source) the codes needed to select drives under your controller and then invent a way to turn "A,B,C, or D" into whatever codes you need. If you need to select sides on a double-sided drive, duplicate the top part of the CCS option then display a "Which Side?" message, translate the response into the proper code, and send it to the proper port.

In the "Read Sector Routine," we again have to separate dma controllers from non-dma controllers. With a dma controller, just issue the go command and check to see when the status register says all the data are transferred. With a programmed I/O controller, you'll have to tell the CPU every step to take to get the data into memory. Under "read1" in the CCS option, I loaded HL with the memory destination of the first byte and initialized a loop counter. Each loop through the data transfer code reads four bytes, so to read an 8-inch single-density sector takes 32 (20H) loops. Double-density 8-inch sectors require 64 (40H) loops. This routine can probably be adapted from the disk read routine in your BIOS. You could call the BIOS routine, but that would make your DDD program nonportable.

"Program Disk Controller" originally output a series of commands to a dma controller, reading them in serial form from the table that followed. If you have a dma controller that needs to be "primed" with a string of commands, create your own conditional table and adjust "Bytes count" for the number of bytes you need to send. Your manual should explain the

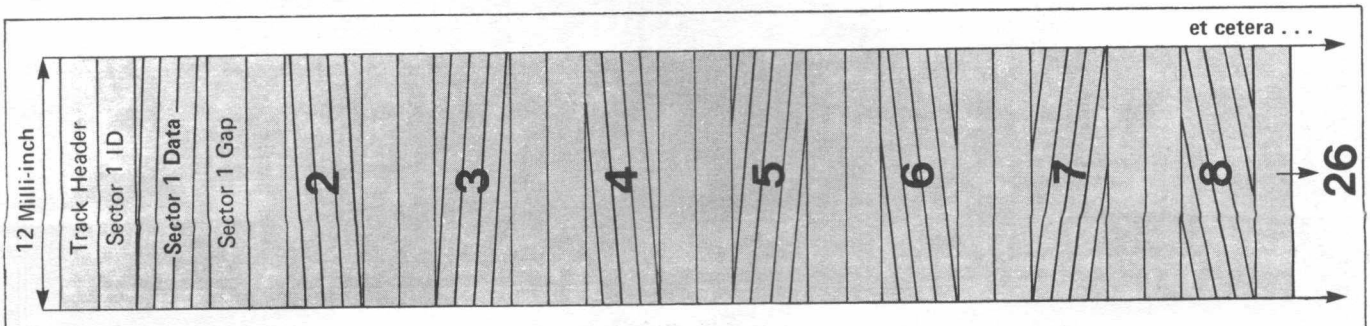


Figure 4.

An azimuth rotation track.

Data sectors, sector IDs, sector gaps and track headers take the same number of bytes as in Figure 2. Azimuth rotation is *highly exaggerated!* Actual rotation is from 18 to 42 minutes – less than a degree. Again, the track is drawn straight for convenience.

procedure; otherwise, find a data sheet on the actual chip used or copy from your BIOS. Even a non-dma controller needs some priming. Mine will look at memory address "dma" (remember "Floppy Controller Port Equates"?) and expect to find the address of "secbuf," a memory buffer to move data to. The other item needed to prime it for a disk read is the setting of "autowait," a function of the CCS 2422 controller card that puts my CPU into a wait state from the time it sends a disk read command until the first byte of data is ready. The catch is that to set "autowait" I have to write to the same port that does disk select; thus I save the bits I used to select the current disk and sent them here with a double function.

"Delay" has already been dealt with (or fudged), so unless you'd like to customize some of the console displays or messages, we've reached the end of the source code! Run it through your assembler and clean up any trivial errors, and the program should be yours. To make it a useful tool, you'll need to create a "before" picture of your drives' operation. Run all the tests on all your drives and write down the results. There are no "correct" results for these tests; however, you'd like to be able to read as many sectors as possible and to have your readings symmetrical around the center of a properly aligned track. If you record a baseline while your drives are working, you'll be able to identify any subsequent changes as possible causes of new malfunctions.

Finally, be aware that flexible diskettes, including the DDD, change size with temperature and humidity. A change of 18°F can move the outer tracks of an 8-inch disk up to 1 milli-inch. On an extremely humid day, the outer tracks may "grow" another 3 milli-inches. If your drives are in a poorly ventilated enclosure where the temperature rises to 125°F, that alone can move traks 7 milli-inches out of alignment. Run your baseline tests under normal conditions of operation after your system is well warmed up, and be sure the DDD has had time to acclimate to the room and to the temperature inside the drive you're testing. Proper drive alignment once the drives are packed into your enclosure may be considerably different from alignment in the open on a test bench.

■■J

(Listing begins below)

Diagnostic Diskette Listing (Text begins on page 40)

```
 1:
 2: ;        DIGITAL DIAGNOSTIC DISKETTE tm  APPLICATION PROGRAM
 3:
 4: ;        BY LOREN AMELANG, BOX 24, PHILO, CA     95466 0024
 5:
 6: ;        THIS CODE EXCERPTED FROM VERSION 1.1   (DDD11.ASM)
 7:
 8: ;        (THE EXCERPT DOES NOT INCLUDE STEPPER HYSTERESIS
 9: ;                                OR INDEX TIMING CHECKS...)
10:
11: ;        This document contains information developed by
12: ;        Dysan Corporation (Dysan) and is furnished for
13: ;        information only.
14:
15: ;        Dysan makes no warranty or representation (expressed
16: ;        or implied) with respect to the accuracy, completeness
17: ;        or usefulness of the information contained herein.
18: ;        Further Dysan assumes no responsibility for liability
19: ;        or damage of any kind which may result from the use
20: ;        of the information contained herein.
21:
22: ;        THIS PROGRAM IS IN THE PUBLIC DOMAIN, AND MAY BE
23: ;                        FREELY COPIED AND DISTRIBUTED
24: ;
25: ;
26: ;        ************************************************************
27: ;
28: ;        USAGE:
29: ;
30: ;        To use this program, a digital alignment disk is required.
31: ;        You may purchase the required "Digital Diagnostic Diskette" (tm)
32: ;        directly from Dysan Corporation.  Available DDD(tm) models are:
33: ;
34: ;        SIDES:.....SINGLE SIDED.....DOUBLE SIDED
```

(Continued on page 50)

Diagnostic Diskette Listing (Listing continued, text begins on page 40)

```
;       DENSITY:.....SINGLE(FM) DOUBLE(MFM)...SINGLE(FM) DOUBLE(MFM)
;       SIZE & TPI:
;       8"      48 TPI   808-100   808-200        808-300   808-400
;       5.25"   48 TPI   508-100   508-200        508-300   508-400
;       5.25"   96 TPI   506-100   506-200        506-300   506-400
;       5.25"  100 TPI   CALL FOR AVAILABILITY    CALL FOR AVAILABILITY
;
;
;       PRICES:  Single Sided - $30.00     Double Sided - $40.00
;                Plus appropriate sales taxes and shipping costs
;                Prices and models subject to change without notice
;
;
;       ORDERING:  Contact your local Dysan sales Rep. or call
;                  Inside Northern California:  (408) 988-3472
;                  Outside Northern California: (800) 551-9000
;
;
;       TECHNICAL INFORMATION:  For information regarding DDD(tm)
;                               applications contact Dysan's
;                               CE Division TECH-LINE: (408)734-1624
;
;
;       ************************************************************************
;
;       PROGRAM SOURCE CODE:
;

wboot   equ     0               ; warm boot address

bdos    equ     5               ; jump address for bdos calls


**********************
*** "CONDITIONALS" ***
**********************
;
;       Choose the appropriate assemble time options for your system,
;       or use this space to add new system options.
;
true    equ     0ffffh
false   equ     not true
;
Dysan   equ     false   ;set this true for Dysan controller
CCS     equ     true    ;set this true for CCS 2422 controller
                        ;       programmed i/o transfer, etc.
lineclr equ     false   ;set this true if your console has
                        ;       a clear-to-end-of-line command
Hazel   equ     false   ;set this true for Hazeltine 1500 terminal
                        ;       (requiring "lead-in" to commands)
Loren   equ     true    ;set this true for Loren's homebrew terminal...
;
;ADD CONDITIONAL OPTIONS FOR YOUR ENVIRONMENT HERE...
;
;


****************************
*** "CONSOLE FUNCTIONS" ***
****************************

;       The following are direct console commands
;       which can be altered for most consoles.

;       note:
;            The text print routine must be altered
;            if "Row" is sent before "Column" or
;            bias is needed.
```

```
104:            if      Hazel           ;version 1.0
105: add$cur    equ     17              ; address cursor command
106: clr$co     equ     28              ; clear to foreground spaces
107: lead$in    equ     126             ; console lead-in
108: end$ln     equ     15              ; clear to end of line
109:            endif
110:
111:            if      Loren           ;version 1.1
112: add$cur    equ     01              ; address cursor command
113: clr$co     equ     05              ; clear to foreground spaces
114:            endif
115:
116: cr         equ     0dh             ; carriage return
117: lf         equ     0ah             ; line feed
118: esc        equ     1Bh             ; escape
119: bs         equ     8               ; back space
120: bell       equ     7               ; audio alert
121:
122:
123: ****************************************
124: *** "FLOPPY CONTROLLER PORT EQUATES" ***
125: ****************************************
126:
127: ;       Change the following Port assignments for
128: ;       Your system.
129: ;
130: ;       "FDC"  Acronym  Floppy Disk Controller
131: ;
132: ;       FDC port assignments:
133: ;
134:            if      Dysan           ;version 1.0
135: fdc        equ     0f8H            ; Base port address FDC
136: ;
137: cmd        equ     fdc             ; command reg.
138: stat       equ     fdc             ; status reg.
139: trk        equ     fdc+1           ; track reg.
140: sec        equ     fdc+2           ; sector reg.
141: data       equ     fdc+3           ; data reg.
142: dma        equ     fdc+4           ; Dma controller
143: dsel       equ     fdc+5           ; drive select
144:            endif
145:
146:            if      CCS             ;version 1.1
147: fdc        equ     30H             ; Base port address FDC
148: ;
149: cmd        equ     fdc             ; command reg.
150: stat       equ     fdc             ; status reg.
151: trk        equ     fdc+1           ; track reg.
152: sec        equ     fdc+2           ; sector reg.
153: data       equ     fdc+3           ; data reg.
154: dma        equ     004CH           ;memory address of pointer to
155: ;                                                 sector buffer
156: dsel       equ     fdc+4           ; drive select port:
157:                                    ;output BX for single density
158:                                    ;       FX for double density
159:                                    ;   where X = 1 for drive A
160:                                    ;             2 for drive B
161:                                    ;             4 for drive C
162:                                    ;             8 for drive D
163:            endif
164:
165: ************************************
166: *** "FLOPPY CONTROLLER COMMANDS" ***
167: ************************************
168:
169: ;       These are common to most systems using the 1793.
170: ;       sdma is a Dysan controller command, ignored in V. 1.1.
171:
172: clear      equ     0d0h            ; clear FDC
173: sdma       equ     87h             ; Start Dma Transfer
174: rsec       equ     80h             ; read sector
175: raddr      equ     0c0h            ; read address
```

(Continued on next page)

Diagnostic Diskette Listing (Listing continued, text begins on page 40)

```
176: seek    equ     18h             ; seek track
177: restore equ     8               ; home Head(s)
178: sdelay  equ     4               ; seek delay flag
179:
180: ************************************
181: *** "FLOPPY CONTROLLER VARIABLES" ***
182: ************************************
183:
184: rate    equ     1               ; step rate
185: retry   equ     1               ; read retries
186: fdelay  equ     11      ; FDC delay
187:                         ; see "delay" routine - this value is for
188:                         ; 4 Mhz. Z-80;  adjust here or in "delay"
189:                         ; to suit your clock and processor...
190: indxbyt equ     2       ; byte in status reg. when index hole
191:                         ; is passing sensor
192: busybyt equ     1       ; byte in status reg. when controller
193:                         ; is busy
194: drqbyt  equ     2       ; byte in status reg. when data is
195:                         ; waiting to be read by the computer
196:
197: ******************************
198: *** "DDD DISKETTE EQUATES" ***
199: ******************************
200:
201: ;       Adjust to match your diskette size and format.
202:
203: ttrk    equ     76              ; total number of tracks
204: sectors equ     26              ; sectors per track
205:                                 ; 26 for 8" DDD, 16 for 5.25" DDD
206: lneg    equ     sectors+2               ; last neg. sector +2
207: lpos    equ     sectors+1               ; last pos. sector +2
208: tsec    equ     (sectors/2)-1   ; centering test
209:
210:         if      Dysan
211: ref     equ     798             ; reference time (single Den)
212:                                 ; (used in "index 7")
213:         endif
214:
215:         if      CCS
216: ref     equ     24eh            ; experimentally determined...
217:         endif
218:
219:
220:
221: ****************************
222: *** "MAIN PROGRAM LOOP" ***
223: ****************************
224:
225:         org     100h
226: ;       ====    ====
227:         lxi     sp,stack        ; init stack
228:         call    select          ; select drive
229: ;
230: main:
231:         lxi     sp,stack        ; reset stack
232:         call    cls             ; clear display
233:         lxi     h,mel           ; display Menu..
234:         call    text
235:         call    ci              ; selection?
```

```
236:            cpi     'R'             ; "Radial"
237:            jz      radial
238:            cpi     'A'             ; "Azimuth"
239:            jz      azimuth
240:            cpi     'C'             ; "Centering"
241:            jz      center
242:            cpi     'H'             ; "Hysteresis"
243: ;          jz      hyster
244:            cpi     'S'             ; "RPM"
245:            jz      rpm
246:            cpi     'I'             ; "Index"
247: ;          jz      index
248:            cpi     'D'             ;"Drive Select"
249:            jnz     ex
250:            call    select
251:            jmp     main            ;new drive selected - set option
252:
253: ex:        cpi     'E'             ; "Exit to DOS"
254:            jz      exit
255:            mvi     a,bell          ; Not valid
256:            call    co
257:            jmp     main            ; try again...
258:
259:
260: ************************
261: *** " EXIT TO DOS" ***
262: ************************
263:
264: exit:
265:            call    cls             ; clear display
266:            jmp     wboot
267:
268: ***************************
269: *** "CONSOLE  STATUS" ***
270: ***************************
271:
272: ;          returns with zero status flag set.
273:
274: cstat:
275:            push    h               ; save all registers
276:            push    d
277:            push    b
278:            mvi     c,11            ; BDOS console status
279:            call    bdos            ; function call added for V1.1
280:            pop     b
281:            pop     d
282:            pop     h
283:            ora     a
284:            ret
285:
286: ************************
287: *** "CONSOLE INPUT" ***
288: ************************
289: ;
290: ;          Returns character in "A" reg.
291: ci:
292:            push    h               ; Save all registers
293:            push    d
294:            push    b
295:            mvi     c,1             ; BDOS console input
296:            call    bdos            ; function call added for V1.1
297:            pop     b
298:            pop     d
299:            pop     h
300:            cpi     esc             ; Exit?
301:            jz      main            ; YES!
302:            cpi     'a'             ; convert to upper case
303:            rc                      ; only...
304:            cpi     '{'
```

(Continued on next page)

Diagnostic Diskette Listing (Listing continued, text begins on page 40)

```
305:            rnc
306:            ani     '^'+1
307:            ret
308:
309:
310:    ************************
311:    *** "CONSOLE OUTPUT" ***
312:    ************************
313:    ;
314:    ;       Enter with character in "A" reg.
315:
316:    co:
317:            push    psw
318:            push    h               ; save all registers
319:            push    d
320:            push    b
321:            mov     e,a             ; Pass char in "E" reg for V1.1
322:            mvi     c,6             ; BDOS direct console i/o
323:            call    bdos            ; function call added for V1.1
324:            pop     b
325:            pop     d
326:            pop     h
327:            pop     psw
328:            ret
329:
330:
331:    ********************************
332:    *** "INPUT TRACK LOCATIONS" ***
333:    ********************************
334:
335:    ;       Input track locations for Radial
336:    ;       Hysteresis,Centering and Index tests.
337:    ;       "B" contents Upper Limit upon entry.
338:
339:    inp$trk:
340:            call    cls             ; clear display...
341:            call    text
342:    ;
343:    inp$trk1:
344:            call    ci              ; waiting...
345:            cmp     b               ; within limits?
346:            jnc     inp$trk2        ; no...
347:            sui     'A'
348:            jc      inp$trk2        ; try again
349:            lhld    temp            ; table pointer
350:            mvi     d,0
351:            mov     e,a
352:            dad     d               ; add offset
353:            mov     a,m             ; load track
354:            jmp     track           ; seek to track
355:    ;
356:    inp$trk2:
357:            mvi     a,bell          ; alert
358:            call    co
359:            jmp     inp$trk1        ; try again...
360:
361:    ************************************************
362:    *** "RADIAL AND HYSTERESIS TRACK LOCATIONS" ***
363:    ************************************************
364:
365:    rh$tbl
```

```
366:          db      0
367:          db      3
368: ;        Note:  some Dysan spec sheets list track 5 rather
369: ;        than 3 as an index track - my Revision A DDD definitely
370: ;        uses track 3...  LA.
371:          db      38
372:          db      41
373:          db      70
374:          db      73
375:
376: ****************************
377: *** "INDEX TRACK TABLE" ***
378: ****************************
379:
380: ix$tbl:
381:          db      0
382:          db      76
383:
384: ************************
385: ** "CENTERING TRACKS" ***
386: ************************
387:
388: cen$tbl:
389:          db      35
390:          db      44
391:          db      47
392:
393:
394: ********************************
395: *** "PRINT DECIMAL NUMBERS" ***
396: ********************************
397: ;
398: ;        Enter with number to be printed in
399: ;        'DE' reg.
400:
401: pnum:
402:          push    h                       ; Setup Stack
403:          lxi     h,10                    ; Terminator...
404:          push    h
405:          push    h
406: ;
407: pn1:
408:          call    div16                   ; Divide Number By (10)
409:          mov     a,d                     ; Check result
410:          ora     e                       ; = 0
411:          jz      pn2
412:          xthl
413:          dcr     l
414:          push    h
415:          lxi     h,10
416:          jmp     pn1                     ; Divide Again
417: ;
418: pn2:
419:          pop     d                       ; Display Values
420:          mov     e,l
421: ;
422: pn3:
423:          mov     a,e                     ; Check for Terminator (10)
424:          cpi     10
425:          pop     d                       ; Next digit
426:          rz
```

(Continued on next page)

Diagnostic Diskette Listing

(Listing continued, text begins on page 40)

```
427:          adi     '0'             ; Add ascii Bias
428:          call    co
429:          jmp     rn3
430:
431:
432: ******************************
433: *** "16 BIT SUBTRACTION" ***
434: ******************************
435: ;
436: ;       Subtract "DE" from "HL"
437: subde:
438:          mov     a,l
439:          sub     e
440:          mov     l,a
441:          mov     a,h
442:          sbb     d
443:          mov     h,a
444:          ret
445:
446:
447: ******************************
448: *** "COMPARE "DE" TO "HL" ***
449: ******************************
450:
451: dehl:
452:          mov     a,l             ; compare "E" to "L"
453:          cmp     e               ; If not zero return
454:          rnz
455:          mov     a,h
456:          cmp     d
457:          ret
458:
459:
460:
461: ************************************
462: *** "16/24 BIT DIVIDE ROUTINE" ***
463: ************************************
464:
465: ;       Divide "HL" by "DE"
466: ;
467:
468: div16:
469:          xra     a               ; optional 16 bit divide
470: ;
471: div24:
472:          sta     msb             ; Normal 24 bit divide
473:          shld    msb1
474:          lxi     h,msb2
475:          mvi     m,24+1
476:          lxi     b,0
477:          push    b
478: div25:
479:          mov     a,e
480:          ral
481:          mov     e,a
482:          mov     a,d
483:          ral
484:          mov     d,a
485:          lda     msb
486:          ral
487:          sta     msb
488:          dcr     m
489:          pop     h
490:          rz
```

```
491:          mvi     a,0
492:          aci     0
493:          dad     h
494:          mov     b,h
495:          add     l
496:          lhld    msb1
497:          sub     l
498:          mov     c,a
499:          mov     a,b
500:          sbb     h
501:          mov     b,a
502:          push    b
503:          jnc     div26
504:          dad     b
505:          xthl
506: ;
507: div26:
508:          lxi     h,msb2
509:          cmc
510:          jmp     div25
511:
512:
513:
514: ********************************
515: *** "CLEAR CONSOLE DISLAY" ***
516: ********************************
517:
518: ;        Clear console to foreground spaces.
519:
520: cls:
521:          if      Hazel
522:          mvi     a,lead$in       ; lead in
523:          call    co
524:          endif
525:
526:          mvi     a,clr$co    ; clear to foreground spaces
527:          jmp     co
528:
529: ******************************
530: *** "CLEAR DISPLAY LINE" ***
531: ******************************
532:
533: cline:
534:          if      lineclr
535:          mvi     a,0             ; starting column
536:          sta     column
537:          lxi     h,set$cur       ; set cursor
538:          call    text
539:          endif
540:
541:          if      Hazel AND lineclr
542:          mvi     a,lead$in       ; lead in byte
543:          call    co
544:          endif
545:
546:          if      lineclr
547:          mvi     a,end$ln        ; clear to end of line
548:          call    co
549:          endif
550:
551:          if      NOT lineclr
552:          mvi     a,11            ; starting column
553:          sta     column
554:          lxi     h,set$cur       ; set cursor
```

(Continued on next page)

Diagnostic Diskette Listing (Listing continued, text begins on page 40)

```
555:          call     text
556:          lxi      h,blanks        ;address of a string of 56 blanks...
557:          call     text
558:          endif
559:
560:          ret
561:
562: ***************************
563: *** "FATAL ERROR MSG" ***
564: ***************************
565:
566: fatal:
567:          if       lineclr
568:          call     cline           ; clear line
569:          endif
570:
571:          mvi      a,30            ; column position
572:          sta      column
573:          lxi      h,set$cur
574:          call     text
575:          lxi      h,me2           ; Fatal error msg
576:          call     text
577:          ret
578:
579:
580: ********************************
581: *** "PRINT STRING OF TEXT" ***
582: ********************************
583:
584: ;        Enter with "HL" pointing to text string,
585: ;        terminate string with Null.
586: ;
587: ;        Direction cursor position is used for displays.
588: ;        The value (-1) at the start of string indicates
589: ;        to this routine
590: ;        that column and row position will follow.
591: ;        This format does not necessarily go out to
592: ;        your terminal.    If your terminal needs row sent before column,
593: ;        or a constant bias added (0,0 is not the upper left
594: ;        corner of the screen), you'll have to invent code
595: ;        within this routine to accomplish that.
596: ;
597: ;        This is the internal representation format:
598: ;             db -1,50,10,'NOW IS THE TIME'
599: ;                 ^  ^  ^
600: ;                 |  |  |_Row position (add bias if needed)
601: ;                 |  |_Column position (add bias if needed)
602: ;                 |_Flag (send console lead-in sequence.
603:
604: text:
605:          mov      a,m             ; load char.
606:          ora      a               ; end of string?
607:          rz                       ; yes
608:          cpi      (-1) AND 0FFH   ; position cursor?
609:                                   ; "and offh" added for ASM compatibility
610:          jz       text1           ; yes
611:          inx      h               ; next char.
612:          call     co              ; output
613:          jmp      text
614: ;
615: text1:
616:          inx      h               ; move passed command
617:
618:          if       Hazel
619:          mvi      a,lead$in       ; leading for console
```

```
620:          call    co              ; issue.
621:          endif
622:
623:          mvi     a,add$cur       ; address cursor
624:          call    co
625:          mov     a,m             ; column position...
626:          inx     h
627:          call    co
628:          mov     a,m             ; row position
629:          call    co
630:          inx     h
631:          jmp     text
632:
633: ************************
634: *** "CENTERING CHECK" ***
635: ************************
636:
637: ;
638: ;       Centering test requires the FDC to read
639: ;       all the sectors before diskette centering
640: ;       is consider OK.
641:
642: center:
643:          lxi     h,cen$tbl       ; track table
644:          shld    temp
645:          lxi     h,me9           ; tracks with
646:          mvi     b,'D'           ; limit
647:          call    inp$trk         ; seek track
648:          call    cls             ; clear display
649:          lxi     h,me10          ; centering msg.
650:          call    text
651:          lxi     h,me15          ; frame
652:          call    text
653:          lxi     h,me11          ; New track msg
654:          call    text
655: ;
656: center1:
657:          call    rd$trk          ; read centering track
658:          mvi     a,10            ; set row position
659:          sta     row
660:          jc      center3         ; fatal error..
661:
662:          if      lineclr
663:          call    cline           ; clear line
664:          endif
665:
666:          mvi     a,30            ; set cursor...
667:          sta     column          ; column position
668:          lxi     h,set$cur
669:          call    text
670:          lda     diff            ; must = 0
671:          ora     a
672:          lxi     h,me3           ; "RE-CLAMP DISKETTE"
673:          jnz     center2
674:          mov     a,c
675:          cpi     tsec            ; all sec read?
676:          jnz     center2
677:          lxi     h,me4           ; "CENTERING OK"
678: ;
679: center2:
680:          call    text
681:          call    cstat           ; abort?
682:          jz      center1
683:          call    ci
684:          cpi     ' '             ; new track?
```

(Continued on next page)

Diagnostic Diskette Listing

(Listing continued, text begins on page 40)

```
685:            jnz     center1         ; no...
686:            jmp     center          ; yes..
687: ;
688: center3:
689:            call    fatal           ; fatal error
690:            jmp     center2
691:
692:
693: ***********************************
694: *** "AZIMUTH ALIGNMENT CHECK" ***
695: ***********************************
696:
697: azimuth:
698:            call    cls             ; clear display...
699:            mvi     a,76            ; seek to azimuth track
700:            call    track           ; move...
701:            lxi     h,me12          ; type of test msg
702:            call    text
703:            lxi     h,me15          ; frame
704:            call    text
705: ;
706: azimuth1:
707:            lxi     h,azi$tab       ; set translation table
708:            shld    xpoint          ; pointers
709:            shld    ypoint
710:            call    rd$trk          ; read azimuth track
711:            mvi     a,10            ; set row position
712:            sta     row
713:            jc      azimuth3        ; fatal error...
714:
715:            if      lineclr
716:            call    cline           ; clear line
717:            endif
718:
719:            mvi     a,20            ; Display negative angle
720:            sta     column
721:            lxi     h,set$cur       ; set cursor
722:            call    text
723:            mvi     a,'-'           ; direction of angle
724:            call    co              ; output...
725:            mvi     a,' '           ; space
726:            call    co
727:            lhld    ypoint          ; translation
728:            mov     e,m             ; data negative...
729:            mvi     d,0
730:            call    pnum            ; display
731:            lxi     h,me5           ; print minutes
732:            call    text
733:            mvi     a,45            ; Display Positive angle
734:            sta     column
735:            lxi     h,set$cur
736:            call    text
737:            mvi     a,'+'           ; direction of angle
738:            call    co              ; output...
739:            mvi     a,' '           ; space
740:            call    co
741:            lhld    xpoint          ; translation
742:            mov     e,m             ; data positive..
743:            mvi     d,0
744:            call    pnum            ; display
745:            lxi     h,me5           ; print minutes
746:            call    text
747: ;
```

```
748: azimuth2:
749:          call     cstat            ; abort?
750:          jz       azimuth1
751:          call     ci               ; test...
752:          jmp      azimuth1         ; no..
753: ;
754: ;        Fatal error Has occurred
755: ;
756: azimuth3:
757:          call     fatal            ; fatal read error
758:          jmp      azimuth2
759:
760:
761: ********************************
762: *** "RADIAL ALIGNMENT CHECK" ***
763: ********************************
764:
765:
766: radial:
767:          lxi      h,rh$tbl         ; table pointer
768:          shld     temp
769:          lxi      h,me7            ; track display..
770:          mvi      b,'G'            ; limit
771:          call     inp$trk          ; select track and seek
772:          call     cls
773:          lxi      h,me8            ; print scale...
774:          call     text
775:          lxi      h,me11           ; Space bar msg
776:          call     text
777: ;
778: radial1:
779: ;
780:          lxi      h,rad$pos        ; radial positive
781:          shld     xpoint           ; cursor table
782: ;
783:          lxi      h,rad$neg        ; radial negative
784:          shld     ypoint           ; cursor table
785: ;
786:          call     rd$trk           ; read radial track
787:          mvi      a,11             ; set row position
788:          sta      row
789:          jc       radial4          ; Fatal error has occurred
790:          call     cline            ; clear line (1)
791:          lda      row
792:          push     psw
793:          inr      a                ; clear line (2)
794:          sta      row
795:          call     cline
796:          pop      psw              ; reset line position
797:          sta      row
798:          lhld     ypoint           ; negative
799:          mov      a,m
800:          sta      column           ; position cursor
801:          mov      b,a
802:          lxi      h,set$cur
803:          call     text
804:          mvi      a,'^'            ; set pointer
805:          call     co
806:          lhld     xpoint
807:          mov      a,m              ; positive position
808:          sta      column
809:          mov      c,a              ; save cursor position
810:          lxi      h,set$cur        ; set cursor
```

(Continued on next page)

Diagnostic Diskette Listing

(Listing continued, text begins on page 40)

```
811:            call    text
812:            mvi     a,'^'
813:            call    co
814:            mov     a,b
815:            sta     column          ; reset column position
816:            lda     row             ; down one line
817:            inr     a
818:            sta     row
819:
820:
821: ************************************
822: *** "SET POINTERS AND DRAW LINE" ***
823: ************************************
824:
825: ;       used to draw the graphic display of the
826: ;       radial error.
827:
828:            lxi     h,set$cur       ; position cursor
829:            call    text
830:            mov     a,c
831:            sub     b               ; line length
832:            dcr     a               ; less (1)
833:            mov     b,a
834:            mvi     a,'|'           ; pointer...
835:            call    co
836: ;
837: radial2:
838:            mvi     a,'_'           ; jointing line
839:            call    co
840:            dcr     b               ; line length -1
841:            jnz     radial2
842:            mvi     a,'|'
843:            call    co
844: ;
845: radial3:
846:            call    cstat           ; check for abort
847:            jz      radial1         ; read again
848:            call    ci              ; new track?
849:            cpi     ' '
850:            jnz     radial1         ; no...
851:            jmp     radial          ; new track...
852:
853: ******************************
854: *** "FATAL RADIAL ERROR" ***
855: ******************************
856:
857: radial4:
858:
859:            if      lineclr
860:            call    cline           ; clear display
861:            lda     row             ; next line
862:            inr     a
863:            sta     row
864:            call    cline
865:            endif
```

```
866:
867:            call    fatal           ; fatal read error
868:            jmp     radial3         ; abort?
869:
870:
871: ******************************
872: *** "CHECK SPINDLE SPEED" ***
873: ******************************
874: ;
875: ;       All timing is based on 4mhz Z80 Cpu
876: ;
877: rpm:
878:            call    cls             ; clear display
879:            lxi     h,me19          ; test performed
880:            call    text
881:            lxi     h,me15          ; frame...
882:            call    text
883:            mvi     a,10            ; set row
884:            sta     row
885: ;
886: rpm1:
887:            call    rpm2            ; compute time in 100us
                                          increments
888:            shld    temp            ; store count
889:
890:            if      lineclr
891:            call    cline           ; clear line
892:            endif
893:
894:            mvi     a,19            ; set cursor
895:            sta     column
896:            lxi     h,set$cur
897:            call    text
898:            lhld    temp
899:            xchg                    ; convert to milliseconds
900:            lxi     h,10
901:            call    div16
902:            push    h               ; Save remainder
903:            call    pnum
904:            mvi     a,'.'           ; decimal point
905:            call    co
906:            pop     h
907:            xchg
908:            call    pnum            ; fraction
909:            lxi     h,me20          ; milliseconds msg
910:            call    text
911:            mvi     a,50
912:            sta     column
913:            lxi     h,set$cur
914:            call    text
```

(Continued on next page)

Diagnostic Diskette Listing

(Listing continued, text begins on page 40)

```
915:            lhld    temp            ; convert to RPM
916:            mvi     a,9             ; divide By 600,000
917:            lxi     d,27c0h
918:            call    div24
919:            push    h               ; remainder
920:            call    pnum
921:            mvi     a,'.'           ; fraction
922:            call    co
923:            pop     h
924:            xchg
925:            call    pnum
926:            lxi     h,me21          ; rpm msg
927:            call    text
928:            call    cstat           ; abort?
929:            jz      rpm1            ; No..
930:            call    ci              ; Esc key?
931:            jmp     rpm1            ; No..
932:
933:
934: ********************************
935: *** "INDEX TO INDEX TIMING" ***
936: ********************************
937:
938: ;
939: ;       With 4 Mhz. Z-80,
940: ;       returns time in 100 Us increments to caller
941: ;       in "HL" regs.
942:
943: rpm2:
944:            call    clrend          ; Clear pending commands
945:            lxi     h,0             ; Clear Counter
946: rpm3:
947:            in      stat            ; Loop Until Index
948:            ani     2               ; Mask All But Index Bit
949:            jnz     rpm3            ; No Index
950: rpm4:
951:            in      stat            ; if Index...
952:            ani     2               ; Wait for No Index
953:            jz      rpm4            ; In Index Hole
954:
955: ;       Add index hole to count
956: rpm5:
957:            call    rpm7            ; 4.25
958:            in      stat            ; 2.75
959:            ani     2               ; 1.75
960:            jnz     rpm5            ; 2.50
961:                                    ; Total = 11.25
962: ;       Count until  next Index
963:
964: rpm6:
965:            call    rpm7            ; 4.25
966:            in      stat            ; 2.75
967:            ani     2               ; 1.75
968:            jz      rpm6            ; 2.50
969:            ret                     ; total = 11.25
970:
971: ************************
972: *** "RPM TIMING LOOP" ***
973: ************************
974: ;
975: ;       The loop time must be adjusted for 100us
976: ;       increments at your clock speed with your processor.
977: ;
978: ;       note:   "RPM5 or RPM6" loop time must be counted.
979: ;
980:
981: rpm7:
982:            inx     h               ; 1.50
```

```
 983:          mvi     a,22            ; 1.75
 984: ;
 985: rpm8:
 986:          dcr     a               ; 1.00  (22 * 1.00) = 22 Usec
 987:          jnz     rpm8            ; 2.50  (22 * 2.50) = 55 Usec
 988:          inx     d               ; 1.50
 989:          inx     d               ; 1.50
 990:          inx     d               ; 1.50
 991:          inx     d               ; 1.50
 992:          ret                     ; 2.50
 993:                                  ; total = 88.75
 994:
 995:
 996: ************************
 997: *** "READ DDD TRACK" ***
 998: ************************
 999: ;
1000: ;        Att: Main read routine for program.
1001: ;             ==============================
1002: ;
1003: ;        Reads the Positive and Negative offset
1004: ;        sectors and returns pointers to the
1005: ;        translation tables set by caller.
1006: ;
1007:
1008: rd$trk:
1009:          MVI     A,7     ;Upper case code adds a simple display of
1010:                          ;       last sector read (+ & -), giving
1011:                          ;       ? and @ if first sectors fail
1012:                          ;       and A thru Z representing the 26
1013:                          ;       possible sectors on an 8" disk
1014:                          ;       or A thru P for a 5.25" disk
1015:          STA     ROW
1016:          MVI     A,39
1017:          STA     COLUMN
1018:          LXI     H,SET$CUR
1019:          CALL    TEXT
1020:          mvi     c,lpos          ; last sector
1021:          mvi     a,1             ; beginning sector positive
1022:          call    rd$sec          ; read positive offsets
1023:          ADI     3eH     ; translate sector # to alpha code */
1024:          CALL    C0      ; display on console  */
1025:          SUI     3eH     ; return to sector # */
1026:          cpi     1               ; unable to read
1027:          jz      rd$trk2         ; first sec?
1028:          sta     diff            ; store last sector tested
1029:          mvi     c,lneg
1030:          mvi     a,2             ; beginning sector negative
1031:          call    rd$sec          ; read negative offsets
1032:          ADI     3eH
1033:          CALL    C0
1034:          SUI     3eH
1035:          cpi     2               ; fatal error?
1036:          jz      rd$trk2         ; set error flag
1037: ;
1038: ;        Positive sector Translation
1039: ;
1040:          lda     diff
1041:          inr     a               ; adj. sector Num.
1042:          sui     4
1043:          rar                     ; divide by two
1044:          lhld    xpoint          ; translate Table pointer
1045:          mvi     d,0             ; setup for Offset
1046:          mov     e,a
1047:          dad     d               ; add offset
1048:          shld    xpoint          ; set pointer
1049: ;
1050: ;        Negative Sector Translation
1051:
1052:          in      sec             ; last sector
1053:          sui     4
```

(Continued on next page)

Diagnostic Diskette Listing (Listing continued, text begins on page 40)

```
1054:           rar                     ; divide by two
1055:           mov     c,a
1056:           sub     e               ; diff. for Hysteresis
1057:           sta     diff
1058:           mov     e,c             ; offset
1059:           lhld    ypoint          ; translation table pointer
1060:           dad     d               ; offset in table
1061:           shld    ypoint          ; set pointer
1062:           ora     a               ; clear carry
1063:           ret
1064:
1065: ;         Set fatal error flag, caller processes
1066: ;         the error.
1067:
1068: rd$trk2:
1069:           stc                     ; carry = Fatal error
1070:           ret
1071:
1072:
1073:
1074: ********************************************
1075: *** "READ SECTORS BY INCREMENTS OF TWO" ***
1076: ********************************************
1077:
1078: ;
1079: ;         Enter with first sector to be read in 'A' reg.
1080: ;         Returns to caller on Error or Last sector.
1081: ;         note:
1082: ;             Z flag set if good read else Nz
1083: rd$sec:
1084:           call    read            ; read sector
1085:           in      sec             ; current sector
1086:           rnz
1087:           adi     2
1088:           out     sec
1089:           cmp     c               ; last sec?
1090:           rz                      ; yes...
1091:           jmp     rd$sec
1092:
1093: ************************************************************
1094: ***  "THE FOLLOWING ARE HARDWARE DEPENDENT ROUTINES" ***
1095: ************************************************************
1096: ;
1097: ;
1098: ***********************
1099: *** "DRIVE SELECT" ***
1100: ***********************
1101: ;
1102: ;         The drive select routine must be expanded
1103: ;         for selected systems.
1104:
1105: select:
1106:
1107:           if      Dysan
1108:           mvi     a,21h           ; select bits
1109:           out     dsel            ; select drive...
1110:           endif
1111:
1112:
1113:           if      CCS     ;The upper case code performs drive
1114:                           ; selection on the CCS 2422 controller
1115:                           ; for single side double density...
1116:           CALL    CLS     ; clear console */
```

```
1117:           LXI     H,SELMSG
1118:           CALL    TEXT        ; print drive selection request */
1119:           CALL    CI          ; get response */
1120:           CPI     'A'         ; accept A,B,C,D, as responses */
1121:           JZ      SELECT1
1122:           JC      SELECT
1123:           CPI     'B'
1124:           JZ      SELECT1
1125:           CPI     'C'
1126:           JZ      SELECT3
1127:           CPI     'D'
1128:           JZ      SELECT4
1129:           JMP     SELECT  ; repeat if invalid choice */
1130:
1131:
1132: SELECT4:ADI     3           ; make low nybble an 8 */
1133: SELECT3:INR     A           ; make a 4 */
1134: SELECT1:SUI     '@'         ; make B into 2 or A into 1 */
1135:           ORI     0F0H        ; add high nybble: */
1136:                               ; F for double density, B for single
1137:           STA     SELPT       ; store select bits for future use... */
1138:           OUT     DSEL
1139:           MVI     A,40H       ; select side 0...  00 selects side 1... */
1140:           OUT     04          ; secondary control port */
1141:           endif
1142:
1143:           jmp     home                ; home drive
1144:
1145: ********************************
1146: *** "SEEK HEAD(S) TO TRACK" ***
1147: ********************************
1148: ;
1149: ;       Enter with new track location in
1150: ;       'A' register.
1151:
1152: track:
1153:           out     data                ; new track
1154:           call    clrend              ; clear FDC
1155:           mvi     a,seek+rate         ; seek command + step rate
1156: ;
1157: track1:
1158:           out     cmd                 ; issue...
1159:
1160:           if      Dysan
1161:           call    delay               ; delay for FDC
1162:           endif
1163:
1164:           if      CCS         ;This controller seems to need more
1165:                               ; delay - I can't explain why... LA.
1166:           call    delayB              ; delay for FDC
1167:           endif
1168:
1169: ;
1170: track2:
1171:           in      stat                ; busy ?
1172:           rrc                         ; test
1173:           jc      track2              ; still busy...
1174:           mvi     a,sdelay            ; set seek delay bit
1175:           sta     dflag
1176:           in      trk                 ; set current track
1177:           sta     ctrk
1178:           ret
1179:
```

(Continued on next page)

Diagnostic Diskette Listing (Listing continued, text begins on page 40)

```
1180:           if      CCS
1181: DELAYB:   PUSH    B       ; extra delay for CCS controller... */
1182:           LXI     B,2000H
1183:
1184: DELAYC:   DCX     B
1185:           MOV     A,C
1186:           ORA     B
1187:           JNZ     DELAYC
1188:           POP     B
1189:           RET
1190:           endif
1191:
1192:
1193: ****************************************
1194: *** "RESTORE HEAD(S) TO TRACK ZERO" ***
1195: ****************************************
1196:
1197: home:
1198:           call    clrend          ; clear FDC
1199:           mvi     a,restore+rate  ; restore command + step rate
1200:           jmp     track1          ; issue command...
1201:
1202: ******************************
1203: *** "READ SECTOR ROUTINE" ***
1204: ******************************
1205:
1206: ;         Enter this routine with the sector to
1207: ;         be read in the 'A' register.
1208: ;
1209: ;         This routine will keep trying to read
1210: ;         the sector until retry count equals zero.
1211: ;
1212: ;         The 1793 controller has a built-in automatic
1213: ;         four retries to read ID, so the value set in
1214: ;         the equate (RETRY) is equal to [ 4 * retry + 1].
1215:
1216: read:
1217:           out     sec             ; set sector
1218:           mvi     e,retry         ; to read
1219: ;
1220: read1:
1221:           call    clrend          ; clear FDC
1222:           call    prg$dma         ; program "DMA" controller
1223:
1224:           if      CCS
1225:           LHLD    DMA     ; set pointer to memory buffer */
1226:           MVI     B,40H   ; initialize loop counter for I/O */
1227:                           ;transfers: 40 for 8" DD, 20 for 8" SD
1228:           endif
1229:
1230:           lda     dflag           ; seek delay flag
1231:           ori     rsec            ; read sector command
1232:           out     cmd             ; issue..
1233:
1234:           if      Dysan
1235:           mvi     a,sdma          ; start transfer
1236:           out     dma
1237:           endif
1238:
1239:           if      CCS
1240: BREAD:    IN      DATA    ; code to perform I/O transfer of  */
1241:           MOV     M,A     ;       data to memory */
1242:           INX     H
1243:           IN      DATA
1244:           MOV     M,A
1245:           INX     H
1246:           IN      DATA
1247:           MOV     M,A
```

```
1248:           INX     H
1249:           IN      DATA
1250:           MOV     M,A
1251:           INX     H
1252:           DCR     B       ; decrement loop counter... */
1253:           JNZ     BREAD
1254:           endif
1255:
1256:           call    comp            ; transfer completed?
1257: ;
1258: read2:
1259:           xra     a               ; clear seek delay
1260:           sta     dflag
1261:           in      stat            ; good read?
1262:           ora     a
1263:           rz                      ; yes...
1264:           dcr     e               ; count off retries
1265:           jnz     read1           ; try again
1266:           in      stat            ; return status
1267:           ora     a
1268:           ret
1269:
1270:
1271: ****************************************
1272: *** "WAIT FOR END OF DMA TRANSFER" ***
1273: ****************************************
1274:
1275: comp:
1276:           call    delay           ; before each check
1277:           in      stat            ; Completed Transfer?
1278:           rrc
1279:           rnc                     ; YES!
1280:           jmp     comp
1281:
1282:
1283: *********************************
1284: *** "PROGRAM DISK CONTROLLER" ***
1285: *********************************
1286:
1287: prg$dma:
1288:
1289:           if      Dysan
1290:           mvi     b,15            ; Bytes Count
1291:           lxi     h,cmd$tbl       ; Command Table
1292: ;
1293: prg$dma1:
1294:           mov     a,m             ; load
1295:           out     dma             ; write to controller
1296:           inx     h
1297:           dcr     b               ; -1
1298:           jnz     prg$dma1
1299:           endif
1300:
1301:           if      CCS
1302:           LXI     H,SECBUF        ; store address of memory buffer */
1303:           SHLD    DMA     ; in pointer for disk controller */
1304:           LDA     SELPT   ; get drive select bits */
1305:           OUT     DSEL    ; sets autowait for 2422 controller... */
1306:                           ; to ensure synchronization of I/O transfer
1307:           endif
1308:
1309:           ret
1310:
1311: ****************************************
1312: *** "COMMAND TABLE DMA CONTROLLER" ***
1313: ****************************************
1314:
1315: ;       This data is used to program Zilog's
1316: ;       z80 DMA controller.
```

(Continued on next page)

Diagnostic Diskette Listing (Listing continued, text begins on page 40)

```
1317:
1318: cmd$tbl:
1319:
1320:          if      Dysan
1321:          db      0c3h            ; Re-Set DMA Controller
1322:          db      8bh             ; Clear Block Counter
1323:          db      79h             ; Recieve Block
1324:          dw      secbuf          ; Address To Store Block
1325:          dw      256             ; Block Size
1326:          db      14h             ; Define Port (B) Address
1327:          db      28h             ; Define Port (A) Address
1328:          db      85h
1329:          db      data            ; Data Reg FDC
1330:          db      8ah             ; Set DMA Controller Active HIGH
1331:          db      0cfh
1332:          db      3               ; Data  ----> Memory
1333:          db      0cfh
1334:          endif
1335:
1336:
1337: ************************************
1338: *** "CLEAR PENDING COMMANDS FDC" ***
1339: ************************************
1340: ;
1341: ;       Clears the floppy controller of any
1342: ;       pending commands.
1343:
1344: clrend:
1345:          mvi     a,clear         ; clear FDC command
1346:          out     cmd             ; issue...
1347:
1348: ****************************************
1349: *** "FDC DELAY TO PROCESS COMMANDS" ***
1350: ****************************************
1351: ;
1352: ;       This delay must be adjusted for different
1353: ;       clock speeds. 50 usec delay loop.
1354: ;
1355: ;       Note:
1356: ;            If altered, Ajust Index Timing Routine.
1357:
1358: delay:                          ;  4.25
1359:          mvi     a,fdelay        ;  1.75
1360: delay1:
1361:          dcr     a               ; 11.00
1362:          jnz     delay1          ; 27.50
1363:          nop                     ;  1.00
1364:          nop                     ;  1.00
1365:          nop                     ;  1.00
1366:          ret                     ;  2.50
1367:
1368: ********************************
1369: *** "RADIAL TRANSLATE TABLE" ***
1370: ********************************
1371:
1372: ;       Cursor Positioning table for radial
1373: ;       alignment test.
1374:
1375: rad$nes:
1376:          db      38              ; 1  millinch
1377:          db      36              ; 2  "     "
1378:          db      34              ; 3  "     "
1379:          db      32              ; 4  "     "
1380:          db      30              ; 5  "     "
1381:          db      28              ; 6  "     "
1382:          db      26              ; 7  "     "
1383:          db      24              ; 8  "     "
1384:          db      22              ; 9  "     "
```

```
1385:           db      20              ; 10 "    "
1386:           db      17              ; 11 "    "
1387:           db      14              ; 12 "    "
1388:           db      11              ; 13 "    "
1389:
1390:  rad$pos:
1391:           db      40              ; 1  millinch
1392:           db      42              ; 2   "      "
1393:           db      44              ; 3   "      "
1394:           db      46              ; 4   "      "
1395:           db      48              ; 5   "      "
1396:           db      50              ; 6   "      "
1397:           db      52              ; 7   "      "
1398:           db      54              ; 8   "      "
1399:           db      56              ; 9   "      "
1400:           db      58              ; 10  "      "
1401:           db      61              ; 11  "      "
1402:           db      64              ; 12  "      "
1403:           db      67              ; 13  "      "
1404:
1405:  ************************************
1406:  *** "AZIMUTH TRANSLATE TABLE" ***
1407:  ************************************
1408:
1409:  azi$tab:
1410:           db      18              ; head azmuith angle
1411:           db      20              ; +/- minutes
1412:           db      22
1413:           db      24
1414:           db      26
1415:           db      28
1416:           db      30
1417:           db      32
1418:           db      34
1419:           db      36
1420:           db      38
1421:           db      40
1422:           db      42
1423:
1424:  ****************************
1425:  *** "PROGRAM MESSAGES" ***
1426:  ****************************
1427:
1428:  me1:
1429:
1430:           db      (-1) AND 0FFH,18,5,'(*)================================
1431:           db      (-1) AND 0FFH,18,6,'(*)        DIAGNOSTIC COMMAND MENU
1432:           db      (-1) AND 0FFH,18,7,'(*)================================
1433:           db      (-1) AND 0FFH,18,8,'(*)
1434:           db      (-1) AND 0FFH,18,9,'(*)   R = Radial      H = Hysteresis
1435:           db     (-1) AND 0FFH,18,10,'(*)
1436:           db     (-1) AND 0FFH,18,11,'(*)   A = Azimuth     I = Index Timi
1437:           db     (-1) AND 0FFH,18,12,'(*)
1438:           db     (-1) AND 0FFH,18,13,'(*)   C = Centering   S = Spindle Sp
1439:           db     (-1) AND 0FFH,18,14,'(*)
1440:           db     (-1) AND 0FFH,18,15,'(*)   D = Drive Sel   E = Exit Progr
1441:           db     (-1) AND 0FFH,18,16,'(*)
1442:           db     (-1) AND 0FFH,18,17,'(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)(*)<
1443:           db     (-1) AND 0FFH,18,19,'SELECTION? ....',bs,0
1444:
1445:  me2:     db      'FATAL READ ERROR      ',0
1446:  me3:     db      'RE-CLAMP DISKETTE ....',bs,0
1447:  me4:     db      'CENTERING OK ....     ',bs,0
1448:  me5:     db      '  Minutes            ',0
1449:
1450:  me7:     db      (-1) AND 0FFH,25,2,'<>=========================<>'
1451:           db      (-1) AND 0FFH,25,3,'<>     TRACK SELECTION      <>'
1452:           db      (-1) AND 0FFH,25,4,'<>=========================<>'
```

(Continued on page 78)

Diagnostic Diskette Listing (Listing continued, text begins on page 40)

```
1453:           db      (-1) AND 0FFH,25,5,'<>                                  <>'
1454:           db      (-1) AND 0FFH,25,6,'<>   A = (0)      D = (41)  <>'
1455:           db      (-1) AND 0FFH,25,7,'<>                                  <>'
1456:           db     (-1) AND 0FFH,25, 8,'<>   B = (3)      E = (70)  <>'
1457:           db     (-1) AND 0FFH,25, 9,'<>                                  <>'
1458:           db     (-1) AND 0FFH,25,10,'<>   C = (38)     F = (73)  <>'
1459:           db     (-1) AND 0FFH,25,11,'<>                                  <>'
1460:           db     (-1) AND 0FFH,25,12,'<>==========================<>'
1461:           db     (-1) AND 0FFH,25,13,'<>    <ESC> Cancels Test    <>'
1462:           db     (-1) AND 0FFH,25,14,'<>==========================<>'
1463:           db     (-1) AND 0FFH,25,16,'TRACK? ...',bs,0
1464:
1465: me8:      db      (-1) AND 0FFH,24,5,'---  RADIAL ALIGNMENT CHECK ---'
1466:           db      (-1) AND 0FFH,10,9,'Away',(-1) AND 0FFH,36,9,'Spindle',(
1467:           db      (-1) AND 0FFH,10,10,'13 12 11 10 9 8 7 6 5 4 3 2 1'
1468:           db              '-1 2 3 4 5 6 7 8 9 10 11 12 13',0
1469:
1470: me9:      db      (-1) AND 0FFH,25,2,'<>==========================<>'
1471:           db      (-1) AND 0FFH,25,3,'<>      TRACK SELECTION     <>'
1472:           db      (-1) AND 0FFH,25,4,'<>==========================<>'
1473:           db      (-1) AND 0FFH,25,5,'<>                                  <>'
1474:           db      (-1) AND 0FFH,25,6,'<>    A = (35)   B = (44)   <>'
1475:           db      (-1) AND 0FFH,25,7,'<>                                  <>'
1476:           db      (-1) AND 0FFH,25,8,'<>          C = (47)        <>'
1477:           db      (-1) AND 0FFH,25,9,'<>                                  <>'
1478:           db     (-1) AND 0FFH,25,10,'<>==========================<>'
1479:           db     (-1) AND 0FFH,25,11,'<>   <ESC>  Cancels Test    <>'
1480:           db     (-1) AND 0FFH,25,12,'<>==========================<>'
1481:           db     (-1) AND 0FFH,25,14,'TRACK? ...',bs,0
1482:
1483: me10:     db      (-1) AND 0FFH,23,5,'---  DISKETTE CENTERING CHECK ---',0
1484:
1485: me11:     db      (-1) AND 0FFH,23,14,'Press <SPACE BAR> For New Track',0
1486:
1487: me12:     db      (-1) AND 0FFH,24,5,'--- AZIMUTH ALIGNMENT CHECK ---'
1488:           db      (-1) AND 0FFH,25,14,'<ESC> Will Cancel AZIMUTH Check',0
1489:
1490: me13:     db      (-1) AND 0FFH,25,2,'<>==========================<>'
1491:           db      (-1) AND 0FFH,25,3,'<>      TRACK SELECTION     <>'
1492:           db      (-1) AND 0FFH,25,4,'<>==========================<>'
1493:           db      (-1) AND 0FFH,25,5,'<>                                  <>'
1494:           db      (-1) AND 0FFH,25,6,'<>    A = (0)    B = (76)   <>'
1495:           db      (-1) AND 0FFH,25,7,'<>                                  <>'
1496:           db      (-1) AND 0FFH,25,8,'<>==========================<>'
1497:           db      (-1) AND 0FFH,25,9,'<>    <ESC> Cancels Test    <>'
1498:           db     (-1) AND 0FFH,25,10,'<>==========================<>'
1499:           db     (-1) AND 0FFH,25,12,'TRACK? ...',bs,0
1500:
1501: me15:     db      (-1) AND 0FFH,15,8,'####################################
1502:           db     (-1) AND 0FFH,15,12,'####################################
1503:
1504: me19:     db      (-1) AND 0FFH,27,5,'--- SPINDLE SPEED CHECK ---'
1505:           db      (-1) AND 0FFH,25,14,'<ESC>  Will Cancel RPM Check',0
1506:
1507: me20:     db      '  Milliseconds',0
1508: me21:     db      '  RPM',0
1509:
1510: BLANKS:  DB      '
1511:          DB      '                ',0
1512:
1513: SELMSG:  DB      (-1) AND 0FFH,15,12,'WHICH DRIVE HAS THE DDD - A,B,C,D ?
1514:          DB      (-1) AND 0FFH,25,14,'SELECTION ? ....',0
1515:
1516:
1517: ************************
1518: *** "POSITION CURSOR" ***
1519: ************************
1520:
1521: set$cur db      0ffh
```

```
column  ds      1               ; set column
row     ds      1               ; set row
        db      0               ; terminator

************************************
*** "PROGRAM VARIABLE STORAGE" ***
************************************

temp    ds      2               ; temp storage
diff    ds      1               ;  1st - 2nd read
ctrk    ds      1               ; current track
savtrk  ds      1               ; temp storage Hysteresis test
dflag   ds      1               ; seek delay flag
point   ds      2               ; table pointer
xpoint  ds      2               ; table pointer positive
ypoint  ds      2               ; table pointer negative
xoff    ds      1               ; positive offset
yoff    ds      1               ; negative offset
bais    ds      1               ; bais from test track (Hysteresis)
hyerr   ds      1               ; hysteresis error on first reading
SELPT   DS      1       ;STORES BYTE FOR DISC CONTROL PORT

;
;       Storage area for 16/24 bit divide routine
;
msb     ds      1
msb1    ds      2
msb2    ds      2

inbuf:  db      3               ;"max" byte for BDOS 10
incnt   ds      1               ;"cnt" buffer for BDOS 10
buffer  ds      4               ; console input buffer
secbuf  ds      256             ; sector buffer

        ds      100             ; stack space
stack   equ     $               ; top down...
```

End Listing

Building a Programmable Frequency Synthesizer

by Michael L. Simon

Michael L. Simon, Space-Time Productions, 2053 N. Sheffield, Chicago, Illinois 60614.

Copyright © 1983 Michael L. Simon. All rights reserved.

Permission is granted for non-commercial use only. Any commercial use without permission from the author is prohibited.

The Master Controller Board (MCB), a single-board computer available from Space-Time Productions, Chicago, Illinois, has an 8253 counter/timer on it that is used for a baud rate generator. Since it has two spare counter time channels, I decided to use them to construct a crystal-controlled programmable frequency synthesizer.

Phase-Locked Loop (PLL) Theory

A phase-locked loop is a very useful combination analog and digital device. It enables a digital frequency divider to multiply frequencies exactly. It performs this trick by controlling the frequency and phase of a voltage-controlled oscillator (VCO) in such a way that the VCO is locked into the frequency and phase of a reference oscillator. (See Figure 1, below.)

An example will help to explain the operation. A reference frequency of 1000 Hz is available; it is used as one input to a phase comparator. The other input comes from the output of the VCO through a programmable divider, which is set to divide by four. For the loop to be locked, the output of the programmable divider must be 1000 Hz. If the frequency is low, the phase comparator will generate longer pulses which, when averaged by the low pass filter, will create a higher voltage at the input of the VCO. This will cause the output of the VCO to increase in frequency, maintaining an output of 4000 Hz.

Similarly, if the output of the VCO is above 4000 Hz, the phase comparator will generate shorter pulses. This will reduce the frequency, again restoring the original 4000 Hz. Thus a phase-locked loop and a divide-by-four circuit are able to multiply an input frequency by four. If the input to the reference of the loop has high stability, the output of the loop will also tend to have high stability.

Figure 2 (page 81) shows the circuit diagram of a phase-locked loop.

The 4046 CMOS Phase-Locked Loop Chip

The 4046 is a CMOS phase-locked loop which is fairly inexpensive and operates over a wide frequency range (from less than 1 Hz to over 1 million Hz). The 4046 has a phase detector that is edge sensitive. This is useful when the signals to be synchronized are not perfect square waves. The chip has a second phase detector that can be used to tell if the loop is out of lock. Because the 4046 is a CMOS device with high input impedance, loop filters with high value resistors and low value capacitors are possible. This makes possible low-frequency loop filters built from small, inexpensive capacitors.

The Single-Board Computer

The MCB is a Z80-based, single-board computer with many useful features. Among these features are 72 parallel I/O lines, two serial I/O ports, and two counter/timer chips with a total of eight counter/timers. Also available for the board is a version of the TDL monitor, modified to allow control of the programmable dividers of the 8253. This program makes frequency synthesizer experiments a snap since it has commands that directly control the dividers in the 8253 counter/timer chip.

One channel of the 8253 is used for the on-board baud rate generator. Each channel of the baud rate generator is set up for a different baud rate; three rates

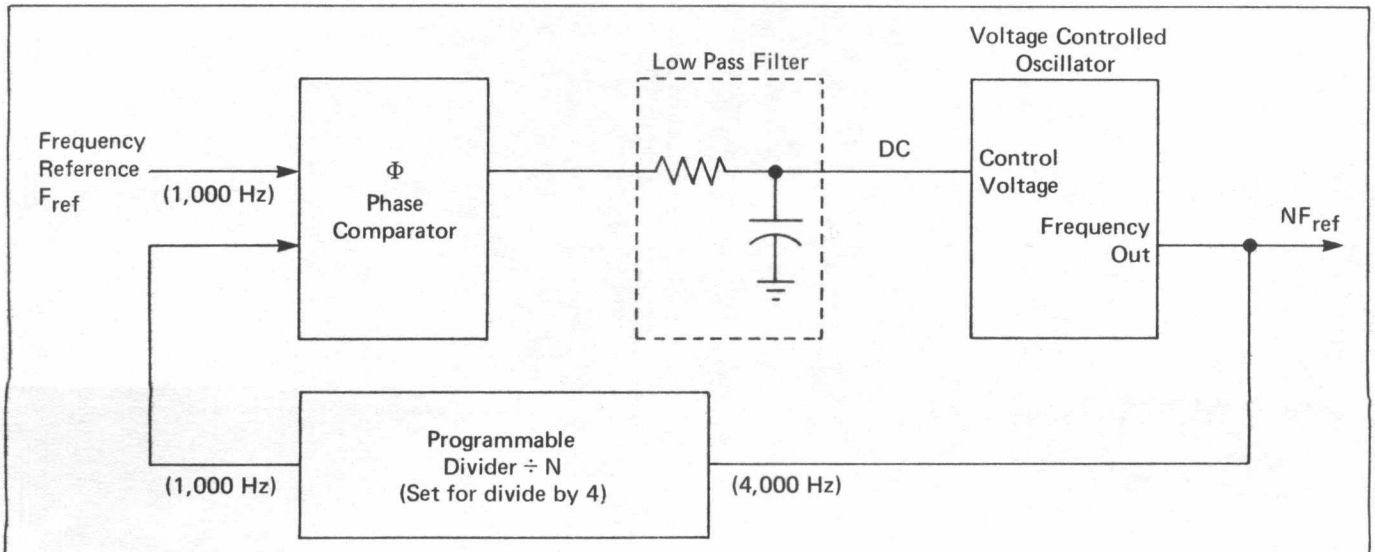


Figure 1.

Phase-locked loop (PLL) example.

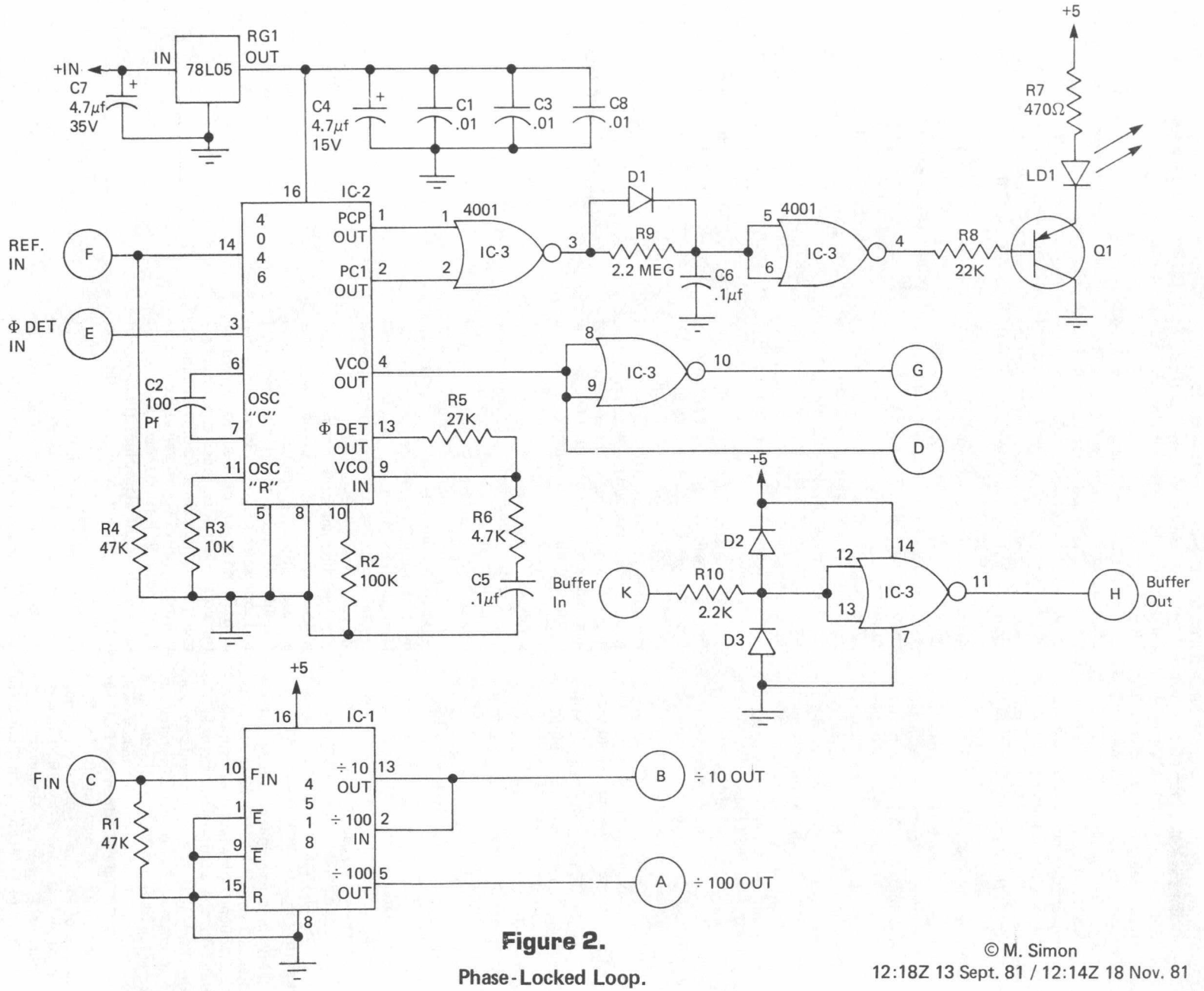


Figure 2.

Phase-Locked Loop.

© M. Simon
12:18Z 13 Sept. 81 / 12:14Z 18 Nov. 81

(110, 300, and 4800) are available. Since I use 300 baud to talk to the MCB, which uses channel 1 of the 8253, channels 0 and 2 are available for experiments. The 8253 is set up by the monitor program to act as a programmable divide-by-N circuit and to produce a square wave. If N is an even number, the output will be a square wave. If N is an odd number, the output will be approximately a square wave with one half of the cycle longer than the other half; refer to Figure 3 (page 82) and Listing One (page 88).

The gate of each channel used must be pulled high for the 8253 to perform the divide function. A 2 MHz buffered, crystal-controlled reference signal is available from the CPU's clock generator. This signal is required by the baud rate generator and is brought to pin 20 of header J-4 from the pads near the crystal by a jumper wire soldered to the underside of the board. From this pin I used reuseable jumper wires on the header pins to bring the 2 MHz signal to the inputs of counters 0, 1, and 2. These jumper wires are about the handiest I have ever used on .025 square pins. They make wrapped wire methods seem very clumsy, especially when doing experiments, since the socketed jumpers are easy to move and reuse.

After hooking up the PLL according to diagram 2, I expected an output frequency of about 45,775 Hz – I say "about" because the output frequency depends on the exact frequency of the CPU time base generator and the divide ratios chosen for the baud rate generators. My oscillator was slightly low, so I got a different value.

I prepared a chart of divide ratios versus hexadecimal numbers. The baud rate generators could be programmed to recognize decimal values. However, by using hexadecimal numbers, over six times the range of divisors is possible.

My first experiment was to get the system to go out of lock on the low end. The frequency at which this happens depends on the components used in the loop filter. With the components chosen, the loop goes out of lock at a reference frequency of about 200 Hz into the phase comparator. The exact frequency will vary with the rate at which the PLL has to change between new inputs. The loop goes out of lock on a jump from divide-by-1000 hex to divide-by-2000 hex on the channel 2 counter. However, if you can inch up to 1F00 and keep the loop in lock, it is possible to jump from there to 2000 hex and still keep the loop in lock.

The same method is used for getting the loop back in lock from an out-of-lock condition. To get the loop to respond to lower frequencies and still stay in lock, try increasing R5 and C5. (Check out *Design of Phase Locked Loop Circuits With Experiments* by Howard M. Berlin, Howard W. Sams Inc., for methods of determining the best values for the loop filter.)

On the high end it is possible to keep the loop in lock until well past 1 MHz. However, I have had some PLL's lock up at the high frequencies. This is probably due to the speed limitation of the chips I used. By removing power and beginning again, everything was returned to normal.

Solid State Scientific of Montgomery, Alabama, makes a version of the 4046 with a VCO guaranteed to oscillate up to 5 MHz. The part is called a 4446. By reducing capacitor C2 to 50 pf, I got mine to stay in lock past 2.5 MHz. For those who want to go past 2.5 MHz, I recommend using about 20 pf at C2. You must also use a divider capable of higher frequency input than the 8253. Fortunately, the Master Controller Board has a socket for the AMD 9513 counter/timer, which is guaranteed up to 7 MHz.

It is possible to generate almost any frequency with the right choice of dividers. For example, I wanted to tune up my AM radio. To do this I needed to tune up the intermediate stages first. This requires a frequency of 455,000 Hz. With the crystal in my board, loading counter 0 with 3A4 hex and counter 2 with 1000 hex gave me a readout on my frequency counter of 454,995.0 Hz. Not bad, but I wanted to get closer.

The first step was to determine the frequency of the clock on the MCB. Checked with my trusty calculator, 3A4 converted to decimal is equal to 932; 1000H is 4096. Dividing 454,995 Hz by 932 found the reference frequency. Multiplied by 4096, the result was the crystal-controlled input clock frequency. My result was 1,999,634.6 Hz.

The dilemma now was to determine which divider ratios would yield the correct output frequency. There is a formula for doing this. Unfortunately, I had long since forgotten it, so I invented a trial-and-error method to determine the correct divisors. Since computers are much faster at trial and error than humans, I wrote a BASIC program (see Listing Two, page 91) to handle it. If I had known the right algorithm, there would have been quicker results, but the program does the job.

The main idea is that the phase-locked loop may be used to generate any desired frequency to any degree of accuracy by choosing the right values for the controlling dividers. By using this program, I came up with values of 808 for divider #0 and 3551 for divider #2. This gives an accuracy of 0.13 ppm. I converted 808 and 3551 to hexadecimal (328h and 0DDFh) and entered the values in the 8253. 455,000.0 Hz was the result!

With suitable dividers following the loop output, audio tones can be generated. These could be used to fine tune your favorite string instrument (see Figure 4, page 82). The 4040 circuit gives output ranging from the lowest to the highest octave, evenly spaced at octave intervals. The even-tempered scale used in most musical instruments uses twelve notes per octave interval. Each note is higher in frequency than the preceding note by a factor of 1.0594631. Table I (below) gives a series of frequencies that facilitates tuning all the notes in any octave. The beauty of this set-up is that with the proper choice of dividers, any scale may be reproduced, regardless of the number of notes or the size of the interval.

The Phase-Locked Loop board is available as a kit for $28.75 postpaid from Space-Time Productions, 2053 N. Sheffield, Chicago, Illinois 60614; (312) 327-0391. The Master Controller Board is available bare for $54.90 with postage. The customized TDL Monitor is available for $69.90. A partially populated MCB, assembled and tested, running Tiny Controller Basic, is $304.90 with postage.

■■J

(Listing begins on page 88)

Note	Ratio	Input to 4040 Divider	A=440
A	1.0000000	450560.00 Hz	440.00000 Hz
A#	1.0594631	477351.69 Hz	466.16375 Hz
B	1.1224620	505736.47 Hz	493.88327 Hz
C	1.1892071	535809.15 Hz	523.25112 Hz
C#	1.2599210	567670.00 Hz	554.36523 Hz
D	1.3348398	601425.42 Hz	587.32951 Hz
D#	1.4142135	637188.03 Hz	622.25393 Hz
E	1.4983071	675077.20 Hz	659.25507 Hz
F	1.5874010	715219.44 Hz	698.45648 Hz
F#	1.6817928	757748.56 Hz	739.98882 Hz
G	1.7817974	802806.63 Hz	783.99084 Hz
G#	1.8877486	850544.00 Hz	830.60937 Hz
A	2.0000000	901120.00 Hz	800.00000 Hz

Table 1
Twelve Tone Scale Frequencies

Frequency Synthesizer (Text begins on page 80)

Listing One

```
; 11:58z 22 MARCH 83 CT FOR PLL ART
CDL's Z80 Macro Assembler C12012-0312
Modified for MASTER CONTROLLER BOARD 21 MARCH 83  M.Simon

                        ; EQUATES FOR THE 8253 TIMER CHANNELS

006C                    CTR0    == 6Ch                  ;8253 CHAN 0
006D                    CTR1    == CTR0+1
006E                    CTR2    == CTR0+2
006F                    CTRTC   == CTR0+3               ;8253 CONTROL
                                                        ; CHANNEL

                        ;THE "BAUD" PROGRAM GETS TWO PARAMETERS OFF THE
                        ; STACK. THE FIRST OFF THE STACK IS A TWO BYTE
                        ; HEX NUMBER THAT IS THE DIVIDER THAT IS TO BE
                        ; USED BY THE COUNTER. THE SECOND NUMBER IS USED
                        ; TO DETERMINE THE COUNTER TO BE USED.

                        ;DIVIDERS CAN BE FROM 2 TO 0FFFFh, COUNTERS
                        ;  ARE 0,1, & 2
                        ;

017A    CD 065E         BAUD:   CALL    EXPR            ;GET 2 PARAMS
                                                        ; & PUT THEM ON
                                                        ;  THE STACK
017D    D1                      POP     D               ;BAUD RATE
017E    E1                      POP     H               ;COUNTER #
017F    7D              BR1:    MOV     A,L             ;TEST FOR CTR #
0180    B7                      ORA     A
0181    2809                    JRZ     OUTC0
0183    3D                      DCR     A
0184    2811                    JRZ     OUTC1
0186    3D                      DCR     A
0187    2819                    JRZ     OUTC2
0189    C2 057F                 JNZ     ERROR

018C    3E36            OUTC0:  MVI     A,00110110B
018E    D36F                    OUT     CTRTC
0190    7B                      MOV     A,E
0191    D36C                    OUT     CTR0
0193    7A                      MOV     A,D
0194    D36C                    OUT     CTR0
0196    C9                      RET

0197    3E76            OUTC1:  MVI     A,01110110B
0199    D36F                    OUT     CTRTC
019B    7B                      MOV     A,E
019C    D36D                    OUT     CTR1
019E    7A                      MOV     A,D
019F    D36D                    OUT     CTR1
01A1    C9                      RET
```

```
01A2    3EB6          OUTC2:  MVI     A,10110110B
01A4    D36F                  OUT     CTRTC
01A6    7B                    MOV     A,E
01A7    D36E                  OUT     CTR2
01A9    7A                    MOV     A,D
01AA    D36E                  OUT     CTR2
01AC    C9                    RET

;THIS PROGRAM DOES THE POWER ON INITIALIZATION OF
; THE 8253 COUNTER TIMERS. THIS PROGRAM IS CALLED
;  BY THE MAIN INITIALIZE PROGRAM. THE THREE
;   COUNTER TIMERS ARE SET UP FOR BAUD RATE
;    GENERATORS. CHAN. #0 = 4800 BAUD
;      CHAN. #1 = 300 BAUD  CHAN. #2 = 110 BAUD

01C8                  CT53IN:
01C8    2E00                  MVI     L,0             ;INIT CTR 0
01CA    11 001A               LXI     D,26.           ; FOR 2Mhz CLOCK,
                                                      ; 4800 BAUD
01CD    CD 017F               CALL    BRI
01D0    2E01                  MVI     L,1             ; CTR 1
01D2    11 01A1               LXI     D,417.          ;  300 BAUD
01D5    CD 017F               CALL    BRI
01D8    2E02                  MVI     L,2             ; CTR 2
01DA    11 0470               LXI     D,1136.         ;  110 BAUD
01DD    CD 017F               CALL    BRI
01E0    C9 .                  RET
```

End Listing One

Listing Two

```
                    PROGRAM #1

10 X=3              ;TIMER 2 DIVIDE NUMBER
20 Y=3              ;TIMER 0 DIVIDE NUMBER
30 PRINT "INPUT CLOCK FREQUENCY":INPUT C
40 PRINT "       OUTPUT FREQUENCY":INPUT O
50 I=O/C              ;RATIO OF OUTPUT TO CLOCK
60 PRINT "INPUT ACCURACY IN PPM":INPUT A
70 A=A/1E6
80 Z= (I-(Y/X))/I
90 IF ABS (Z)<A THEN GOTO 1000
100 IF Z>0 THEN 600
110 IF Z<0 THEN 500
120 GOTO 1000
500 X=X+1:GOTO 80
600 Y=Y+1:GOTO 80
1000 PRINT"TIMER #2 DIVIDER" X,"TIMER #0 DIVIDER"Y
1010 PRINT"ACTUAL ACCURACY" Z
1020 GOTO 10
```

End Listing Two

CP/M EXCHANGE

by Robert Blum

The column's a little short this month due to an accelerated deadline caused by CP/M '83. The interface series that I promised would start this month will be held off until next month. I apologize for the delay but I think you will enjoy it once it starts.

User Areas — Bah, Humbug

Several months ago, I cried on your shoulder about how old and decrepit my system had become. Well, since then, new disk drives have been installed, a new motherboard is now keeping a handle on the bus signals, and a 15MB hard disk happily gobbles up all the data I can feed it. Unfortunately, I should never have installed the hard disk because it pointed out to me one black hole that exists in CP/M: there is no easy way to partition logically related groups of data files.

Let me explain a little further. Right from the beginning I delegated my floppy disk drives to specific tasks. Drive A: is called the runtime drive. It contains only executable (.COM) files and any temporary work files that may be needed by the application programs. Drives B: and beyond are used for data files. I adopted this arrangement because it eliminates any confusion when switching from one application system to another. When I want to run word processing, the WP runtime disk goes into drive A:, and one of various WP data disks is mounted in a drive designated for data files. Very simple, very clean, and easy to manage.

Now comes the puzzle: how to allocate the hard disk so it is able to fit into my established scheme. My first attempt was to split the 15MB into two logical CP/M disks. The first one, drive C:, was 8MB in size with all remaining space given to drive D:. Merrily I copied disk after disk of data onto the Winchester. It's hard to describe my feeling of excitement as the data was gobbled up by an insatiable appetite. However, I became concerned after I ran my first directory listing. Displayed was such a hodge-podge of files that I knew something had to be changed.

I shuddered at my first thought — mapping a number of logical floppies onto the hard disk. Fortunately, it was only a fleeting thought. Even as I write this I can hear somebody scraping their fingernails across a blackboard.

User numbers would partition the files the way I wanted, but at the expense of duplicating a number of programs in each of the user areas. The overhead of multiple copies of the same program is mandated by CP/M 2.2's inability to automatically switch user areas when searching for a file. For example, if you are currently in user area three, any file that is opened must also be in the same user area. This includes program overlays. One solution to the problem would be to use a CCP replacement such as ZCPR, but this would only provide a partial solution because an expanded search would only be available when ZCPR was resident. Once the executable program is loaded it has no way of knowing what user area it came from and hence no way of finding its overlays. Unfortunately, when Digital Research designed user areas, they failed to provide a means by which a file's FCB could communicate to the BDOS which user area it is in.

I don't mean to place all the blame on DR for this inadequacy. They provided a BDOS function that will switch the user area for any application program that is smart enough to use it. Unfortunately, none of the application developers decided to use it. Micro Pro took one step in the right direction by modifying WordStar to look at drive A: for its overlay files if they are not found on the currently logged-in floppy. This is better than nothing, but if the user area is not taken into account, it is of little use. Of course, at the time that CP/M 2.2 became available, disk capacities were relatively low and disks were typically swapped for each application. Today, however, hard disks are common and there is no excuse for having to allocate 15MB of space into logical floppies. As much as I hate to admit it, the standard version of CP/M 2.2 simply isn't suited to the hard disk environment.

It wouldn't seem too difficult to trap all the disk I/O BDOS calls and swap the user area for FCBs that contain a file extension of .COM or overlays that reside in the same user area. One problem with this occurs when an overlay file doesn't use .OVR for its extension. However, a little extra code could provide the desired results.

Converting to CP/M Plus would solve my problem. As a standard feature, it will search user area zero for any file that is not found in the current user area, although there are some restrictions. The target file must be marked with the $SYS attribute, and it cannot be written to. This arrangement works well for executable programs, which is what I am complaining about, but for my money they don't get the prize. In designing CP/M Plus, DR decided to add some very useful features. A file can be password-protected, and time and date stamping is available if your system supports a clock of some sort. To store this extra information, extended directory entries are recorded along with the regular directory entries.

Unfortunately, to cross-access user areas requires that the program manually switch the user area by calling the BDOS. I think a little extra work should have gone into allowing a program to pass an extended FCB address to the BDOS when a file is opened. This extended FCB would contain all the additional information necessary to interface to the new features that I have already mentioned, plus the user number. Better yet, instead of a user number, how about using an 8-character text name like the file name?

Making comparisons between different operating systems is usually like comparing apples to oranges, but I want to point out how files are handled on another system that I use daily. To protect the innocent I will call this new system Alpha. Alpha is a Z80-based computer with 64K of memory. The resident operating system requires approximately 10K, and the only transient part of the system is the operator interface, which is equivalent to CP/M's console command processor. The only place that my comparison begins to break down is that Alpha has a separate Z80-based processor that handles all disk I/O. I think my analysis still holds because CP/M Plus requires a minimum of 96K if you want to use the extended features. In Alpha's case, this would also be enough to hold all the disk logic.

A file specification on Alpha is contained in two control blocks. The first is called a "file control block" and is used for various pointers and general parameter passing. A two-byte area of the FCB points to another control block, the "file name block." Contained in the FNB are all the file's attributes in text form. Included are eight characters for the file set name (equivalent to user area), eight characters for the file name, and eight more characters for the file extension. Following this information is a complete description of the file. This includes record length, read and write passwords, and some other control information which isn't germane to this discussion. When the program wants to open a file, the FCB address is loaded into a register and a command code is placed into another. A call is made to the operating system through a restart

instruction, and away you go.

If it sounds simple, it is. Take, for example, the command line necessary to assemble a program:

```
ASM BLUM-PROGRAM.ASM
```

A few moments later the BLUM-fileset contains the assembler's output even though I was logged into the TEMP-fileset.

I have worked on a number of computers over the years and Alpha is my favorite. The real surprise comes when one considers Alpha's origin. Until Alpha's operating system was far enough along in the development cycle to stand alone, it was run under CP/M. Even after a few years of refinement, there are a number of CP/M's operational features still being used. Of course, I understand that a lot of things can be done when designing a new system that can't be done when modifying an existing one because of compatibility problems. Still, I think a lot can be accomplished if the motivation exists. I am sure that this discussion won't teach DR anything new; I have met a few of their people and they are sharp. But like so much else in this business, decisions are based on market potential. And as long as the semiconductor manufacturers continue to bring out new and improved processors every couple of years, there is little time left to do anything but convert the existing systems. Well, as I have said before, that's the state of the art.

A Point Well Taken

As I was putting the final touches on this month's column, I received a note from a reader complaining about the apparent lack of proofreading of the twittle subroutine that I ran a couple of months ago. He found the word "asterisk" misspelled in the comments describing the program. I must admit that my first impression was that he was being rather nit-picky, but after a moment's thought (it takes that long for my defense mechanism to readjust), I decided that he had a valid point. To the eyes of many, words are like notes to the ears of a musician. I am not an artist, but I am aggravated by a fuzzy radio station or a scratched record; my reader found a misspelled word to be offensive to his eyes. I apologize for the inexcusable error and will pay more attention to detail in the future.

DRI Patch 3 — SUBMIT.COM

Products Affected: CP/M 2.2

Copyright © Digital Research, Inc. 1982. All rights reserved. Used with permission of Digital Research, Inc. Text adapted from the original.

Error Description: If drive A is not the default drive when you run the SUBMIT program, the $$$.SUB file is created on the currently logged-in disk. Therefore, you cannot run a SUBMIT job from any other drive than A. (This occurs because the CCP will only search drive A for the $$$.SUB file.) After you make the changes shown in Listing One (page 94), the system will always create the $$$.SUB file on drive A.

Patch Procedure: Make a back-up copy of SUBMIT.COM before you use DDT to make the changes in Listing One.

SKIPNIF — Last But Not the Least

Last month David Kirkland shared with us his method of speeding up SUBMIT file truncation. When he submitted those valuable comments he also included his version of the original SKIPIF program written by John Ramsey. Dave's program, SKIPNIF, allows one extra parameter on the command line to specify how many records from the SUBMIT file are to be skipped. This is a valuable addition to an already excellent set of SUBMIT file control routines. I think this brings the subject of SUBMIT file processing to a logical conclusion. By using the many routines that have been published in *DDJ* over the past year, you can achieve a substantial level of control over how SUBMIT files are processed. A complete listing of Dave's SKIPNIF program can be found in Listing Two (page 94).

DDJ

(Listings begin on page 94)

CP/M Exchange (Text begins on page 92)

Listing One

```
A>DDT SUBMIT.COM
DDT V2.2
NEXT PC
0600 0100
-D5BB
05BB 00 24 24 24 20  .$$$
05C0 20 20 20 20 53 55 42 00 00 00 1A 1A 1A 1A 1A 1A      SUB...
05D0 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A  ..........

-
-S5BB
05BB 00 1
05BC 24 .
-GO
A>SAVE 5 SUBMIT.COM
```

End Listing One

Listing Two

```
;
;  SKIPNIF  -- conditionally skip n commands within a sumbit file
;
;       written by David Kirkland   6/12/83
;       based on Don Wright's QUITIF (DDJ No. 71, Sept. 1982 at 8, 14).
;
;       revised 6/31/83 to incorporate the "zero" option, based on
;       John C. Ramsey's SKIPIF (DDJ No. 82, Aug. 1983 at 100-07).
;
;       Usage:
;
;       SKIPNIF Ambig    fn    n     skip if fn is ambiguous
;       SKIPNIF Exists   fn    n     skip if fn exists
;       SKIPNIF Missing  fn    n     skip if fn does not exist
;       SKIPNIF Null   [ fn ] n     skip if fn not specified not exist
;       SKIPNIF Zero     fn    n     skip if fn has zero length or does not exist
;
;       Only the first character of the first keyword (the AMBIG or EXISTS or
;       whatever) is checked   If an invalid keyword is entered, the
;       n commands will be unconditionally skipped.
;
;       This code written for Ithaca Intersystems IASM assembler; note
;       that "ccir" is what most people call "cpir", and "ccdr" is "cpdr".
;
tpa     equ     0100h
bdos    equ     0005h
fcb1    equ     005ch           ; first fcb prepared by CCP
name1   equ     fcb1+1          ; start of filename of fcb1
fcb2    equ     006ch
name2   equ     fcb2+1
cmdtail equ     0080h
;
drive   equ     'M'             ; drive on which $$$.SUB file resides
;                               ; ('A' for normal folk)
;  BDOS codes
```

```
;
PrtStr  equ     9
Open    equ     15
Close   equ     16
Srch1st equ     17
Erase   equ     19
Read    equ     20
;
skipif  org     tpa
;
; first parse command line by
; - determining NSkip, the number of lines to skip if condition satisfed
; - ensuring that the contents of fcb2 at 6c contain the fn argument and
;   not the NSkip argument (this problem occurs when the command is
;       skipnif null NSkip
;   i.e., when the fn argument actually is not entered and the skip should
;   be executed).
;
        lxi     h,cmdtail
        mov     c,m             ; length of command line
        mvi     b,0
        dad     b               ; hl points to last character in command line
        inx     h
        mvi     m,' '           ; mark end of command line with space
; skip over "NSkip", the last argument on command line
        call    scanbl          ; scan off trailing blanks
        ccdr                    ; scan backwards until a space reached
        push    h               ; save for later

; now skip back over the next to last arg, which is "fn" if present
        call    scanbl          ; find end of next arg
        ccdr                    ; find start of the arg

; now move back once more.  Normally, there will be one argument left, the
; AMBIG or NULL or whatever.  But in our special case of NULL with no
; fn entered, there will be nothing left.
        call    scanbl          ; throw away extra blanks
        mov     a,b             ; are there any characters unprocessed in the
        ora     c               ; command line? if not, then there were only
        jnz     DoNum           ; two arguments given on the command line

match   mvi     a,' '           ; if only two args, add a blank for the second
        sta     name2           ; arg to make fcb2 null
        jmp     DoNum

; scanbl scans backwards over the command line discarding blanks;
; when a nonblank character is encountered, it stops.
; h -> str, bc = length remaining; return with hl -> first nonblank

scanbl  mov     a,b             ; see if any characters remain
        ora     c
        rz                      ; none left
        mvi     a,' '
scnLp   cmp     m               ; *hl : ' '
        rnz
        dcx     h               ; next char
        dcx     b               ; count down
```

(Continued on next page)

CP/M Exchange (Listing continued, text begins on page 92)

Listing Two

```
        jmp     scanbl
;
; PROCESS DECIMAL NUMBER--modified from ZCPR
;
DoNum
        pop     h
        inx     h
        inx     h               ; hl points to first char in number
; convert the decimal number pointed to by hl into binary and store
; at N2Skip
        mvi     c,0             ;C=ACCUMULATED VALUE set to zero
NUM1:
        MOV     A,M             ;GET CHAR
        CPI     ' '             ;DONE IF ' ', which marks end of number
        JZ      NUM2
        INX     H               ;PT TO NEXT CHAR
        SUI     '0'             ;CONVERT TO BINARY (ASCII 0-9 TO BINARY)
        CPI     10              ;ERROR IF >= 10
        JNC     NUMERR
        MOV     D,A             ;DIGIT IN D
        MOV     A,C             ;NEW VALUE = OLD VALUE * 10
        RLC
        RLC

        RLC
        ADD     C               ;CHECK FOR RANGE ERROR
        JC      NUMERR
        ADD     C               ;CHECK FOR RANGE ERROR
        JC      NUMERR
        ADD     D               ;NEW VALUE = OLD VALUE * 10 + DIGIT
        JC      NUMERR          ;CHECK FOR RANGE ERROR
        MOV     C,A             ;SET NEW VALUE
        jmp     NUM1            ;do next digit
;
;
NUMERR:                         ; come here if number invalid
        mvi     c,PrtStr
        lxi     d,NErrMsg
        call    bdos
        ret
;
                                ; come here once number finished
NUM2:
        MOV     A,C             ;GET ACCUMULATED VALUE
        sta     N2Skip          ; store the value for later

; now actually do what the caller asked for:

        lda     name1           ; first character of the AMBIG or EXISTS or ...
        lxi     h,name2

        cpi     'A'
        jz      ambig
        cpi     'E'
        jz      exists
        cpi     'M'
        jz      missing
        cpi     'N'
        jz      null
```

```
        cpi     'Z'
        jz      zero
                                ; if none of the above, fall through and skip

; skip over next N2Skip commands in $$$.SUB
skip    mvi     c,Open          ; open $$$.SUB
        lxi     d,subfile
        call    bdos
        inr     a               ; check if $$$.SUB exists
        rz                      ; if not, return

        lxi     h,subfile+14
        mvi     m,0             ; zero the S2 byte of the fcb to force a write
                                ; of the modified fcb back to the directory
        lda     N2Skip
        mov     c,a
        inx     h
        mov     a,m             ; decrement the number of records in $$$.SUB
        sub     c               ; by N2Skip
        jp      notneg
        xra     a               ; if skip by more than in file, use zero
notneg  mov     m,a

        mvi     c,Close         ; close $$$.SUB to rewrite the modified fcb
        lxi     d,subfile
        call    bdos

        mvi     c,PrtStr        ; tell the user what we've done
        lxi     d,skipmsg
        call    bdos

        xra     a               ; zero a and set flags for zero/missing
        ret

; abort if 2d parm ambiguous
ambig   mvi     a,'?'
        lxi     b,11
        ccir                    ; cpir for people with normal assemblers
        jz      skip
        ret

; abort if file exists
exists  call    search
        jnz     skip
        ret

; abort if file missing
missing call    search
        rnz
        jmp     skip

; abort if 2d parm null
null!   mvi     a,' '
        cmp     m
        rnz
        lxi     h,name2+8
        cmp     m
        rnz
        lda     fcb2
        ora     a
```

Listing Two

```
        rnz                     ; d: given without more
        jmp     skip

; abort if 2d parm is a zero length file
zero    call    missing         ; if file does not exist, its deemed zero length
        rz                      ; if so, missing has skiped N2Skip records in
                                ; $$$.SUB, so now we return to the CCP
        mvi     c,Open
        lxi     d,fcb2
        call    bdos            ; open the specified file

        mvi     c,Read
        lxi     d,fcb2
        call    bdos            ; try to read the first record
        ora     a
        rz                      ; read was OK, so not zero length, don't skip
        jmp     skip            ; read failed, zero length
; search for first to see if any file exists to match fcb
search  mvi     c,Srch1st
        lxi     d,fcb2
        call    bdos
        inr     a               ; adjust flags: Z <-> not found
        ret
;
N2Skip  db      0

skipmsg db      "Next commands skipped",13,10,'$'
NErrMsg db      "Illegal number to skip",13,10,'$'
subfile db      drive+1-'A',"$$$     SUB",0,0,0,0
;
```

End Listing

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

Version 2.0 of MS-DOS was enhanced with some 40 new function calls for use by application programs. These provide support for the hierarchical directories, device-independent I/O, and memory allocation. Most of them are rather radical departures from the CP/M-ish origins of MS-DOS and show a clear tendency toward providing a Unix-like environment.

In the process of preparing a PC-DOS programmer's reference book for publication next year, I have been trying each of the new functions out in turn by writing small assembly language utilities. When I reached function 4BH, called EXEC, I was especially intrigued. EXEC is designed to allow an application program (which we may call the "parent") to load and execute any other program from the disk, then regain control when the "child" is finished. The parent can pass a considerable amount of information to the child in the form of a command line, default File Control Blocks, and a set of strings called the "environment." Similarly, the child can pass an exit code back to the parent when it finishes its work.

The child program can in turn load other programs, and so on through many levels of control until you run out of memory. This function has plenty of uses even in a single-tasking, single-user operating system. For example, you could write your own screen-oriented, menu-driven command processor which would become the user's interface – one would never need to learn the more complex commands like MODE, CHDIR, and PATH (the same thing could be accomplished with a batch file but it would run much slower). Alternatively, you could load a second copy of the native Command Line Processor (COMMAND.COM), and let it run under the control of your program, rather than directly on top of MS-DOS.

In a multi-tasking environment such as we expect to see in MS-DOS 3.0, this function will be even more useful and will undoubtedly be elaborated with several additional features. Under such an operating system, a parent task can "spawn" any number of child tasks, which can execute concurrently and asynchronously, and communicate by means of queues, semaphores, and pipes.

Well, you say, pie in the sky is all very nice, but why is the EXEC function getting so much attention in this magazine column? The answer is, of course, that when I tried to actually use the function I ran into any number of glitches and hazy spots in the documentation.

The "official" definition of the EXEC function, slightly boiled down, is given in Figure 1 (below). It looks straightforward enough, if you ignore the problem of what format the parameter block's DWORD addresses might be in. The documentation notes that "when your program received control, all available memory was allocated to it. You must free some memory (see call 4AH) before EXEC can load the program you are invoking. Normally, you would shrink down to the minimum amount of memory you need, and free the rest."

The manual goes on to state that all open files of the parent are duplicated in the newly created child task. Also inherited from the parent is some new creature called an "environment block," which is rather vaguely defined as a series of ASCII strings in the form "NAME=parameter."

Each of the strings in the environment block is terminated by a zero byte, and the whole set of strings is terminated by an additional zero byte. The environment always starts on a paragraph boundary, and its segment address is ultimately placed at offset 2CH of the Program Segment Prefix. You can either leave that slot in your EXEC parameter block as zero (in which case the child will acquire the

Call with:

AH	=	4BH
AL	=	load type (see below)
DS	=	segment of ASCIIZ path & program name string
DX	=	offset of ASCIIZ path & program name string
ES	=	segment of parameter block
BX	=	offset of parameter block

Returns:

Carry Flag set if unsuccessful call. AL = error code.

Carry Flag cleared if successful call. Except for CS and IP, all other registers, including the stack pointers, are destroyed.

Load type = 0 : Load and execute program. "A program segment prefix is established for the program and the terminate and control/Break addresses are set to the instruction after the EXEC call."

Parameter block format:

WORD	segment address of environment block
DWORD	pointer to command line to be placed at new Program Segment Prefix + 80H
DWORD	pointer to default FCB #1, to be passed at new Program Segment Prefix + 5CH
DWORD	pointer to default FCB #2, to be passed at new Program Segment Prefix + 6 CH

Load type = 3 : Load overlay. Does not create a Program Segment Prefix, and does not begin execution of the overlay.

Parameter block format:

WORD	segment address where file will be loaded
WORD	relocation factor to be applied to the image

Figure 1.

Specification of arguments and results of MS-DOS EXEC function 4BH.

same environment block as your calling program), or you can provide a pointer to your own set of strings. The manual nonchalantly adds that the maximum size of the environment block is 32 Kbytes.

This zinger introduces a whole new set of mysteries. What in the world could the environment block possibly contain that could be as large as 32 Kbytes? Furthermore, the manual explicity states that the environment block for any given program is static, implying that if more than one generation of child programs is resident in RAM, each one might have a distinct and separate copy of these potentially 32 Kbyte rascals floating around somewhere. My goodness!

Paging through the PC-DOS manual in search of further enlightenment, I found some help in appendix E which stated: "the environment built by the command processor (and passed to all programs it invokes) will contain a COMSPEC string at a minimum (the parameter on COMSPEC is the path used by DOS to locate COMMAND.COM on disk). The last PATH and PROMPT commands will also be in the environment, along with any environment strings entered through the SET command."

Thus, the environment block contains at most three strings that actually mean something to MS-DOS (PATH=, COMSPEC=, and PROMPT=). Any other strings found in the block by an application program were placed there for its information by a SET command issued by the operator or by another application program and have no effect on the behavior of the operating system proper. Loading a dummy COM file under the control of DEBUG, getting the pointer from offset 2CH in the Program Segment Prefix, and dumping out memory, I found that the "default" environment in my system is indeed only the two strings "PATH=" and "COMSPEC=A:\COMMAND.COM." Not so formidable after all.

The major remaining puzzle turned out to be the SETBLOCK function (MS-DOS function 4AH), which your program is supposed to use to "shrink down" the amount of space allocated to it before using EXEC to load another program. The manual says, "on entry, ES contains the segment of the block, BX contains the new requested block size in paragraphs." But when the "block" refers to the application program itself, and not to some piece of RAM it requested later, what segment to use: the segment of the Program Segment Prefix or the base of the Code Segment? Or does "block" actually refer to some kind of memory descriptor parameter block buried in COMMAND.COM?

Attempts to test this out with the Debugger were completely baffling, every attempt resulting in an error code. It turns out that since the Debugger essentially "owns" all of memory and loads your program via its own mechanisms, a SETBLOCK call by a program running under the control of DEBUG is illegal by definition – from the operating system's point of view, it is trying to modify a memory allocation that does not "belong" to it. In any event, the "segment of the block" that should be fed to function 4AH is the Program Segment Prefix.

Finding the number of paragraphs to "shrink" your program's memory allocation down to is also a little tricky, at least for EXE files. You might think that you can simply make an Equate to an address called Program_End, put that in a data segment at the end of your program, and load that segment address into BX at execution time with the aid of the GROUP modifier, thereby letting the Linker and Loader do all the work.

Unfortunately, that method won't work – at least not all the time. The Linker moves all the declared segments around to suit its own purposes, and the physical order of segments in your source code doesn't usually correspond to the order of segments in the resultant EXE file. The best way I have figured out to get the total size of the program at runtime is to put an equate to "Program_End" into another module, which is separately assembled, and reference it with an EXTRN in your main program. At link time, specify the module containing the "Program_End" equate as the last in the list of OBJ files.

Of course, you can also "cheat" by simply assembling and linking the program, looking at the size of the EXE file in the directory listing, and then reassembling the program with the correct size in the code as a literal value. This assumes that all data areas used by the program are embedded as declared constants and variables. An approach like this could cause you trouble later if you modify the program in a way that increases its size, then forget to update the literal too.

To keep the amount of painful interaction between the EXEC function and *DDJ* readers to a minimum, I am publishing herewith a very detailed example program named FXN4BH (see the listing, below), using EXEC to load the MS-DOS utility CHKDSK.COM, that is guaranteed to work. It requires that both FXN4BH.EXE and CHKDSK.COM be present on the "current" or default disk drive. FXN4BH can fail if there is not enough room above it in RAM to load and execute CHKDSK.COM, or if the transient portion of COMMAND.COM in highest RAM (which contains the actual loader) has been destroyed and there is not enough free memory to reload it.

■■J

16-Bit Toolbox Listing (Text begins on page 102)

```
1                                       name    FXN4BH
2                                       page    55,132
3                                       title   'FXN4BH --- demo PC-DOS EXEC function'
4                                   ;
5                                   ; FXN4BH --- demonstrate use of the
6                                   ; PC-DOS 2.0 EXEC function call 4BH
7                                   ;
8                                   ; Copyright (c) 1983 by Ray Duncan
9                                   ;
10      = 000D                      cr      equ     0dh     ;ASCII carriage return
11      = 000A                      lf      equ     0ah     ;ASCII line feed
12                                                          ;
13      0000                        cseg    segment para public 'CODE'
14                                                          ;
15                                          assume  cs:cseg,ds:data,ss:stack
16                                                          ;
17      0000                        demo    proc    far
18                                                          ;at entry DS & ES = PSP
19      0000  1E                            push    ds      ;Save address for final
20      0001  33 C0                         xor     ax,ax   ;FAR RET to PC-DOS on stack
21      0003  50                            push    ax
22                                                          ;save copy of SS:SP for use
23                                                          ;after return from overlay
24      0004  2E: 8C 16 0050 R              mov     cs:STK_SEG,ss
25      0009  2E: 89 26 0052 R              mov     cs:STK_PTR,sp
26                                                          ;
27                                                          ;Reserve 1000H bytes for
28                                                          ;this loader and release
29                                                          ;the rest of memory for
30                                                          ;use by the overlayed program.
31      000E  BB 0100                       mov     bx,100h ;ES=segment of PSP of loader
```

(Continued on page 106)

16-Bit Toolbox Listing (Listing continued, text begins on page 102)

```
0011  B4 4A                        mov     ah,4ah    ;BX=paragraphs to reserve
0013  CD 21                        int     21h
                                              ;make the messages in data
                                              ;segment addressable
0015  B8  ---- R                   mov     ax,seg DATA
0018  8E D8                        mov     ds,ax
001A  8E C0                        mov     es,ax
                                              ;jump if memory
                                              ;de-allocation failed
001C  72 2A                        jc      ALLOC_ERR
                                              ;print memory successfully
                                              ;released
001E  BA 001E R                    mov     dx,offset MSG2
0021  B4 09                        mov     ah,9
0023  CD 21                        int     21h
                                              ;
                                              ;now load and execute
                                              ;the overlaid program.
0025  BA 00A4 R                    mov     dx,offset PGM_NAME
0028  BB 00B0 R                    mov     bx,offset PAR_BLK
002B  B0 00                        mov     al,0
002D  B4 4B                        mov     ah,4bh
002F  CD 21                        int     21h
                                              ;restore stack pointers
                                              ;to state before EXEC call
0031  2E: 8E 16 0050 R             mov     ss,cs:STK_SEG
0036  2E: 8B 26 0052 R             mov     sp,cs:STK_PTR
                                              ;Make data segment
                                              ;addressable again
003B  B8  ---- R                   mov     ax,seg DATA
003E  8E D8                        mov     ds,ax
                                              ;print message that loader
                                              ;successfully regained control
0040  BA 0059 R                    mov     dx,offset MSG3
0043  B4 09                        mov     ah,9
0045  CD 21                        int     21h
                                              ;now exit to PC-DOS
0047  CB                          ret
0048                        alloc_err:        ;come here if memory
                                              ;cannot be released
0048  BA 0000 R                          mov     dx,offset MSG1
004B  B4 09                            mov     ah,9
004D  CD 21                            int     21h     ;print error message and
004F  CB                               ret             ;exit to PC-DOS
                                                       ;
0050                        demo       endp
                                                       ;
                                                       ;these two variables must
                                                       ;reside in Code Segment so
                                                       ;that they are addressable
                                                       ;after return from overlay.
```

```
0050  0000                    stk_seg  dw       0          ;original SS contents
0052  0000                    stk_ptr  dw       0          ;original SP contents
                                                           ;
0054                          cseg     ends
                                                           ;declare a stack area
                                                           ;for use by this loader
0000                          stack    segment  para stack 'STACK'
                                                           ;allow 64 bytes in this case
0000    40 [                          db       64 dup (?)
              ??
                  ]
0040                          stack    ends
                                                           ;declare data segment to
                                                           ;contain variables and tables
0000                          data     segment  para public 'DATA'
                              ;
0000  0D 0A                   msg1     db       cr,lf
0002  55 6E 61 62 6C 65               db       'Unable to release memory.'
      20 74 6F 20 72 65
      6C 65 61 73 65 20
      6D 65 6D 6F 72 79
      2E
001B  0D 0A 24                        db       cr,lf,'$'
001E  0D 0A                   msg2     db       cr,lf
0020  4D 65 6D 6F 72 79               db       'Memory above loader released.'
      20 61 62 6F 76 65
      20 6C 6F 61 64 65
      72 20 72 65 6C 65
      61 73 65 64 2E
003D  0D 0A 4E 6F 77 20               db       cr,lf,'Now loading CHKDSK.COM.'
      6C 6F 61 64 69 6E
      67 20 43 48 4B 44
      53 4B 2E 43 4F 4D
      2E
0056  0D 0A 24                        db       cr,lf,'$'
0059  0D 0A                   msg3     db       cr,lf
005B  4C 6F 61 64 65 72               db       'Loader regained control from CHKDSK,'
      20 72 65 67 61 69
      6E 65 64 20 63 6F
      6E 74 72 6F 6C 20
      66 72 6F 6D 20 43
      48 4B 44 53 4B 2C
007F  0D 0A                           db       cr,lf
0081  6E 6F 77 20 6D 61               db       'now making final exit to PC-DOS.'
      6B 69 6E 67 20 66
```

(Continued on next page)

16-Bit Toolbox Listing (Listing continued, text begins on page 102)

```
132                 69 6E 61 6C 20 65
133                 78 69 74 20 74 6F
134                 20 50 43 2D 44 4F
135                 53 2E
136     00A1  0D 0A 24                          db      cr,lf,'$'
137                                     ;
138                                                     ;drive, path, and name of program
139                                                     ;to be loaded and executed.
140     00A4  5C 43 48 4B 44 53         pgm_name db     '\CHKDSK.COM',0
141           4B 2E 43 4F 4D 00
142                                     ;
143     00B0  ---- R                    par_blk  dw     ENVIR   ;segment address of
144                                                     ;environment descriptor
145                                                     ;
146                                                     ;full address of command line
147                                                     ;to be passed at offset 80H
148     00B2  00BE R                            dw      offset CMD_LINE     ;in overlaid
149     00B4  ---- R                            dw      seg CMD_LINE        ;program's PSP
150                                                     ;
151                                                     ;full address of default
152                                                     ;File Control Block to be
153                                                     ;passed at offset 5CH in
154     00B6  00C5 R                            dw      offset FCB1         ;overlaid
155     00B8  ---- R                            dw      seg FCB1            ;program's PSP
156                                                     ;
157                                                     ;full address of default
158                                                     ;File Control Block to be
159                                                     ;passed at offset 6CH in
160     00BA  00EA R                            dw      offset FCB2         ;overlaid
161     00BC  ---- R                            dw      seg FCB2            ;program's PSP
162                                     ;
163                                                     ;actual command line tail
164                                                     ;to be passed to overlay
165     00BE  04 20 2A 2E 2A 0D         cmd_line db     4,' *.*',cr,0
166           00
167                                     ;
168                                                     ;first default FCB to
169     00C5  00                        fcb1     db     0       ;be passed to overlay
170     00C6     0B [                           db      11 dup ('?')
171                   3F
172                      ]
173
174     00D1     19 [                           db      25 dup (0)
175                   00
176                      ]
177
178
179                                                     ;second default FCB to
180     00EA  00                        fcb2     db     0       ;be passed to overlay
181     00EB     0B [                           db      11 dup (' ')
182                   20
183                      ]
184
```

```
00F6    19 [                    db      25 dup (0)
                00
                    ]
                        ;
010F                    data    ends
                                        ;declare separate data
                                        ;segment to contain
                                        ;environment descriptor
0000                    envir   segment para 'ENVIR'
                                        ;
                                        ;Search path used by PC-DOS
                                        ;to look for commands or
                                        ;batch files not found in
0000  50 41 54 48 3D 00         db      'PATH=',0 ;the current directory
                                        ;
                                        ;Search path used by PC-DOS
                                        ;to locate COMMAND.COM
0006  43 4F 4D 53 50 45         db      'COMSPEC=A:\COMMAND.COM',0
      43 3D 41 3A 5C 43
      4F 4D 4D 41 4E 44
      2E 43 4F 4D 00
001D  00                        db      0       ;extra 0 byte designates
                                        ;end of environment
001E                    envir   ends

                        end     demo
```

End Listing

TRANSFORM

Company: MasterComputing, Inc., P.O. Box 17442, Greenville, SC 29606
Price: $39.95
Circle Reader Service No. 121
Reviewed by Michael Favitta

MasterComputing's (MCI) TRANSFORM is a language preprocessor for Microsoft BASIC and other compatible BASIC's. It translates MCI's enhanced BASIC syntax (which includes block structured statements) into Microsoft BASIC 5.X syntax. The purpose of TRANSFORM is to allow BASIC programmers to develop programs that are easier to debug, understand, and maintain, by using a structured source language.

TRANSFORM is designed for CP/M systems that have 48K of memory. Translated programs will run under MBASIC 5.X or can be compiled with BASCOM 5.3. The translator is distributed on an 8-inch single-density disk along with a sample program and a utility program named SIFT.

A Structured BASIC Language

MCI has made three changes to Microsoft BASIC to create an enhanced version of BASIC. MCI has added blocked control structures, an INCLUDE facility, and replaced line numbers with labels.

To add blocked structures to BASIC, the Microsoft BASIC statements ON-GOTO and IF-THEN-ELSE were modified and a new statement, REPEAT-UNTIL, was added. The syntax of all other statements remains the same. The end-of-block sentinels, ENDGOTO and ENDIF, were added to the standard syntax of the ON-GOTO and IF-THEN-ELSE statements. The translator implements the three additional statements by replacing MCI's blocked structures with combinations of standard Microsoft IF-THEN and GOTO statements. This produces an intermediate object program that is standard Microsoft BASIC. The two examples in *Table I* (at right) show how the IF-THEN-ELSE-ENDIF and REPEAT-UNTIL statements are translated.

Line numbers are replaced with labels in the TRANSFORM source language. A label is any combination of alphanumeric characters up to 255 characters in length that is preceded by the character "@". The translator generates the line numbers that are required by Microsoft BASIC when the intermediate object file is created. A symbol table can be produced that contains the label to line number mapping. A restriction: only comments may follow a label definition, since the translator turns all label definitions into REMarks for documentation in the intermediate object program.

MCI added an INCLUDE facility to allow any file that contains BASIC statements to be copied into a program. A DEFINE statement may be used in conjunction with an INCLUDE statement to redefine the variable names in the included source. This allows a programmer to use the same procedure in many programs, or several times in a single program, without modifying the variables used in the procedure. Since all BASIC variables are global, using a facility such as MCI's INCLUDE is as close as you can get to having local procedures in BASIC. The INCLUDE statement may be nested up to five levels deep.

TRANSFORM allows a programmer to enable and disable the translator from within the program. Two special labels, @PREON and @PREOFF, are used for this purpose. When the translator is disabled, the standard Microsoft BASIC ON-GOTO and IF-THEN-ELSE statements can be used. @PREON and @PREOFF function correctly with the INCLUDE statement by only affecting the level of INCLUDE at which they were specified.

The TRANSFORM Translator

The TRANSFORM translator consists of two COM files, TF.COM and TF1.COM. To invoke the translator you need only enter TF. The program prompts the user for all information that is needed to perform the translation. Most options have a default response that can be chosen by entering a carriage return. Default file name extensions of PRE and BAS are assumed for the source input and object files. TRANSFORM searches for the source if it is not on the specified drive. The order of the search is the currently logged-in drive, followed by the system drive. TRANSFORM's processing options include line number and increment selection for the translated source, listing and error message routing, hardcopy formatting information, and symbol table creation.

Once file name and option selection is complete, TRANSFORM begins its first pass on the source file. This pass translates the structured BASIC source program to standard Microsoft BASIC. The

TRANSFORM Source	Translated Object Code
IF (condition) THEN	1000 IF NOT (condition) THEN 1200
.	.
(block of statements)	(block of statements)
.	.
ELSE	1190 GOTO 1400
	1200 REM
.	.
(block of statements)	(block of statements)
.	.
ENDIF	1400 REM
REPEAT	1000 REM
.	.
(block of statements)	(block of statements)
.	.
UNTIL (condition)	1200 IF NOT (condition) THEN 1000

Table 1
Examples of how the IF-THEN-ELSE-ENDIF and REPEAT-UNTIL are translated.

translator may be interrupted at any time. Once interrupted, you may continue, translate a different source file, or terminate the translator. At the completion of pass one, TRF.COM is loaded and pass two begins. This pass performs all the line numbering that is required by Microsoft BASIC. When the translator has finished, an intermediate object program has been created which can be loaded by MBASIC or compiled by BASCOM.

TRANSFORM programs must be created with a text editor, since BASIC editors always require line numbers. Some text editors, such as Wordstar, set the high order bit on certain characters. This causes problems for the translator, so a utility named SIFT is provided to strip off all the high order bits in a text file. This utility is not needed if your text editor doesn't set the high order bit on any ASCII characters.

Documentation

The documentation that is provided by MCI is divided into two sections, a tutorial and a user's manual. The tutorial demonstrates how the translator is invoked and describes the procedures that are needed to get from MCI-enhanced BASIC to Microsoft BASIC. An additional feature in the tutorial is a discussion of structured programming and how it can be used to simplify the task of creating an "error-free" program. As an example, the tutorial proceeds step by step through the design of a program to format and print BASIC program source files. The source code for the example program is included, so that the user may experiment with TRANSFORM without first having to write a program. Besides being the basis for the tutorial, the print program is a useful utility.

The user's manual is a logical progression of information about TRANSFORM. The first part of the manual contains general information on the purpose of TRANSFORM and its theory of operation. The second part details the syntax of all aspects of MCI's enhanced BASIC language.

Reviewer's Comments

TRANSFORM allows a programmer to write BASIC programs using blocked control structures that have been present in other high-level languages for years. Ideally, there would be no use for such a product, as people who wished to write structured programs would use an available high-level language. Unfortunately, due to the number of years Microsoft BASIC has been around, the availability of a compatible compiler, and the costs associated with purchasing several language processors, this product does serve a useful purpose. Many CP/M software products also require Microsoft BASIC, such as MCI's MCDISPLAY, making the use of a structured language impossible.

One disadvantage to MCI's approach to structured BASIC is that it adds another step in the process of creating an executable program. It is a costly step regarding compile time, as translating a 500-line program takes between eight and ten minutes. There is also potential for problems in the translated source due to the way TRANSFORM implements its blocked structures. In Microsoft BASIC any non-zero expression is considered to represent the true condition for an IF statement. MCI uses "IF NOT(condition) THEN" as part of the translation for each of the blocked structures. This creates the restriction that all conditionals used by the blocked control structures must evaluate to true (minus one) or false (zero). Although this restriction is well-documented, the burden is on the user to make sure that it is followed.

MCI's product TRANSFORM does work. The documentation is complete and the translator is easy to use. If you already have Microsoft BASIC, it would be difficult to find a structured language interpreter or compiler of this caliber for $39.95.

COMMX-PAC

Company: Hawkeye Grafix, 23914 Mobile, Canoga Park, CA 91307
Computer System: CP/M 2.0 or greater
Price: COMMX-PAC: $150, or COMMX only: $99
Circle Reader Service No. 123
Reviewed by Gene Head

I write for several computer magazines and have a listed phone number. So I suppose I get more than my normal share of phone calls from individuals with questions about microcomputers. Eight out of ten callers are requesting help getting a modem working. Seven of those eight are non-programming types trying to get some version of Ward Christensen's MODEM program working. For many, it is a simple matter to set a program's equates to a serial RS-232 status and data port and to monitor the proper status bits For just as many this task is impossible.

Will Pierce at Hawkeye Grafix has taken on the task of delivering a modem communications and control package that is factory patched to run without change on the user's computer system. When ordering COMMX-PAC, the user specifies his/her computer and modem, and Hawkeye does the rest. The purchaser receives a disk with programs that run without the need for any adaptations. Well almost. . . .

The package I ordered for my IMSAI MIO serial I/O card was delivered with the status bit logic reversed. In other words, when executing a status check, the instruction JZ (jump if zero) should have been JNZ (jump if not zero). I know the MIO card is working properly and wired per the manufacturer's specifications because the firmware monitor in a factory ROM controls the MIO card without a hitch. Now it should be noted that IMSAI has been out of business for several years and expecting a vendor to support equipment which is no longer supported by even the manufacturer could be asking too much.

The package my company ordered for a brand-new Molecular Super Micro 30 had two fatal bugs. Molecular's N/Star operating system uses an expanded and larger-than-normal CCP. Fortunately, this package was purchased as assembly language source code. I had no trouble tracking down three places in the source code that expected the normal sized CCP and making the proper modifications. The second error surfaced when the "phone is ringing" response from the smart modem caused the activity time-out timer to reset. This was fixed by telling the smart modem not to send responses after a caller connected.

COMMX-PAC includes, in addition to COMMX and CONSOLX (discussed below), a "data base management" program and a message board program both written in Hawkeye Grafix compiled VBASIC. Neither of these programs performs as well as the public domain programs available on larger RCPM systems and through user's groups.

COMMX-PAC is two separate packages: COMMX for connecting to a host computer like the Source or an RCP/M and CONSOLX for making your computer act like a host or RCP/M.

COMMX

COMMX is the terminal program that allows your computer to communicate with other (host) computers. COMMX is far superior to even the most advanced public domain MODEM program that I have seen to date. It can virtually communicate with any system like IBM TSO, HP-3000, as well as RCP/M's. It has keyboard macro definitions; an entire call, connect, and log-on procedure can be compressed to one or two keystrokes. A half-dozen separate programs allow for an automatic dial-up directory, encoding and decoding private files and un-loading binary files to INTEL HEX format files for systems that are restricted to sending and receiving only seven bits of data.

Hawkeye has included Dick Greenlaw's squeeze and un-squeeze programs among its utilities. These can cut modem connect time by 50% by compacting the

file to be sent, sending the compacted version and then un-compacting it at the receiving end. There is also a sorted disk directory utility but this utility is not as impressive as the public domain program SD-44.COM (Super Directory) described in The CP/M Exchange column of *DDJ*, June 1983.

COMMX supports two error-checking protocols and two methods of file transfer. The COMMX-to-COMMX file transfer system seems to be more advanced than the COMMX-to-MODEM7 system. Earlier versions of COMMX didn't support the Christensen protocol and I'm glad to see the newer version does.

The most advanced feature of COMMX is the E-MAIL feature. Using the dial-up directory, keyboard macros, and a submit file, COMMX can establish communication with other COMMX/CONSOLX sites and exchange mail files while each site is unattended. The entire process can be initiated and then delayed a specified number of hours and minutes to take advantage of off-hour phone charges. COMMX continues trying to establish communications until the job is completed. This E-MAIL requires the most effort to get operational, but the manual is clear and well laid out in describing the E-MAIL features.

The manual also includes an index and even a glossary for non-computing types to refer to. Eighty percent of the manual is dedicated to COMMX with the remaining twenty percent devoted to CONSOLX.

CONSOLX

CONSOLX, like COMMX, is configured at the factory to the user's machine and modem. The list of supported machines and modems is long and growing. I guess nine of ten personal as well as business micros are currently supported.

CONSOLX allows your computer to function as a host to callers accessing your system. Field selectable options include a password, ring-back operation, a welcome message, and UART configuration to either 7-bit even parity or 8-bit no parity. An auto start feature allows your system to execute a program after the user gets past the optional password and welcome message.

At the Money Tree (the company I work for) we run a system that logs callers in, gives them access to proprietary financial information, and logs them off, keeping track of time and charges. Because of its ability to answer the phone and then autoload and execute other programs, CONSOLX works well as the base system. Callers see only as much of the system as they need to see. I know of at least one other business using CONSOLX commercially.

Conclusion

I had to write the routines to switch baud rates in the CONSOLX program and I think these should be included in the package. Also, working without carrier detect requires disorderly log-offs to time out for five minutes before the next caller can connect. I was able to get around this on the Molecular, but I'm not sure this problem can be solved generally for all systems.

For some of us, getting a program to work is half the fun of computing. For others who want it to work the first time and every time, COMMX-PAC from Hawkeye Grafix is clearly worth the $150 price tag. Furthermore, if you have no need for making your computer a host computer and only want to originate calls to other host systems, the COMMX program alone sells for $99.

I commend Hawkeye Grafix for an outstanding package in COMMX-PAC. They have taken on a tough job of providing software for various hardware environments, and as far as I know they are doing the job well. My calls to Will Pierce have always been given professional yet personal treatment. I can personally recommend all their fine products, especially the COMMX-PAC communications package.

■■J

BOOK REVIEWS

An Introduction to Programming and Problem Solving with Pascal, Second Edition

By G. Michael Schneider, Steven W. Weingart, and David Perlman
Published by John Wiley & Sons, Inc.
$21.95, 468 pages
Reviewed by William G. Hutchison, Jr.

"What? *Another* Pascal textbook?" If your reaction was the same as this reviewer's, please do not give up in despair. This book is not another quick-and-dirty attempt to cash in on the demand for computer texts; it is a thoroughly professional effort. It comes from that hotbed of Pascal activity, the University of Minnesota. Andy Mickel read the proof copy. You may remember him as the former editor of *Pascal News*. Niklaus Wirth was interested enough in the project to write a foreword.

This is clearly a college text. The authors wrote it to be used as text for the course CS I, Computer Programming I, of ACM Curriculum '78. They have achieved a very good balance of subject material. Some authors overload their readers with thorny calculus problems to be programmed. This is useful material, but not suitable for an introductory programming course. This text is evenly balanced between text processing applications, commercial (payroll and inventory), games (LIFE), and elementary statistics. The authors use real programs as examples, not meaningless fragments cooked up just to display some feature of the language.

The authors follow their own advice, which is to plan what to do before beginning the coding. Chapter 1 is devoted to problem specification. They point out correctly that most program errors start from errors or ambiguities in the statement of the problem, so it is worthwhile to spend considerable time getting it right at the beginning. Chapter 2 is about developing algorithms in pseudo-code. It contains a brief discussion of program efficiency and asymptotic analysis (e.g., big-O function, n-squared versus $n \log n$ performance), not enough to bog down the reader, but enough to stimulate the student to take the next course.

In addition to the usual programming material, this book includes a chapter on modular program design, and one on program debugging, testing, documentation, and maintenance. Unlike some authors, they do not succumb to the illusion that structured programming means no errors in programs! That is what makes this book more than just another textbook. The technical material is well organized and presented, and it also contains a great deal of practical wisdom, which will save the careful reader from many unnecessary mistakes.

The Pascal language used here is that of the Jensen & Wirth text, *PASCAL Users Manual and Report*, Springer-Verlag, 1974. It is presented in syntax graphs inline in the text as needed, and complete in an appendix. The timing of the text may be unfortunate. It is a rewrite of a 1978 edition. At that time, there was no official standard for Pascal, so it contains no reference to ISO standards or to conformant array parameters. That is not a serious problem for an introductory text, because the core of the language is almost unchanged since 1978. There is practically no reference to UCSD Pascal or other non-standard versions. There is a brief discussion of versions with INTERACTIVE files (which do not do a GET first in READ), and a note about versions with direct access files.

The only thing missing from this excellent text is a discussion of quicksort, which other authors have managed to present in an elementary manner. Let's not produce another generation of programmers who know only bubble sort!

Language Translations

By John Zarrella
Published by Microcomputer Applications, Inc.
$14.95, 200 pages
Reviewed by Michael Carter

Not the least of the advantages of the microcomputer revolution is the encouragement and indeed the incentive it gives the owner of a personal computer to delve into its workings and operations. This will inevitably lead to the spread of computer know-how from a select few to a wider audience – a development which I for one heartily applaud. It also implies the existence of a market for introductory computer texts, of which this book is a good example. The author of such a book faces a considerable problem – how to aim at an easy presentation without oversimplification. The current author is partially – but only partially – successful.

No user of a personal computer can avoid forming an intimate acquaintance with at least one language translator (most commonly BASIC). A language translator is a computer program which accepts a human-oriented source language as input, translating it into a form which can be directly executed by the computer. It is the medium through which the user conveys his or her wishes to the machine.

After a general introduction, Zarrella deals in turn with the three main types of language translator: assemblers, interpreters (e.g., APL, BASIC, Forth), and compilers (e.g., C, COBOL, Pascal, Fortran). He appropriately emphasizes two important points. First, interpreters and compilers are not distinct categories but rather the ends of a continuum with most language translators lying somewhere in-between. The reader will thus understand why a BASIC program is stored in the machine in a form which he or she cannot readily read. The second important point is that the basic tasks required of a language translator are the same whether it is an assembler, compiler, or interpreter. Consequently, assemblers, compilers, and interpreters are made up of essentially the same functional components.

Zarrella makes convenient use of the latter point in the second half of the book when he deals in successive chapters with the functional components of any language translator: lexical analysis, syntax analysis, semantic analysis, code generation, and optimization. The lexical analyser examines the source text and divides it into its component parts: symbols, numbers, punctuation, and so on. The syntax analyser parses the input program to verify that it is a valid program in the language and to provide an appropriate framework for the next step. This is semantic analysis which determines the meaning of the source program. Code generation produces the actual executable machine code. Finally, optimization makes minor changes in the executable machine code to increase execution speed and/or decrease storage requirements. It should not be concluded that these five components of a language translator can necessarily be distinguished as separate entities that follow one another in the order given. They are conceptual divisions, and with the exception of optimization, they must be provided for in every language translator. The five constituent parts provide an appropriate framework for approaching the general task of language translation.

The five chapters dealing with the constituent parts of a language translator

are preceded by a chapter on "Languages and Grammars." This is the theoretical core on which language translation, in particular syntax analysis, is based. Unfortunately, this is not easy material to grasp and the chapter will probably require more than one reading. It is here and in the chapter dealing with syntax analysis that Zarrella's exposition falters. I think this is because he is trying to compress some complex ideas into two short chapters. Unfortunately, this material is really the core of the matter and it left me feeling unsatisfied. I think some more material would have been justified. Further, I found his use of the analogy of a divisionalized company to explain top-down parsing overdone and superfluous.

There are three supplementary chapters which have not yet been mentioned. One deals with program development utilities: linkers, loaders, library managers, etc. A second deals with macros, which are most commonly (but not exclusively) found in conjunction with assemblers. A third deals with the important topic of error processing. A glossary, a short bibliography, and an index round out the book.

This book has an admirable objective: to present the general concepts of language translation, emphasizing the similarities between the different classes of assembler, interpreter, and compiler in a manner accessible to the neophyte. This is a very ambitious task and one in which the author is not entirely successful. I suspect that it is more likely to whet the appetite than satisfy it. Hopefully, it will encourage the reader to seek further information in one of the more substantial texts. To any computer user who is curious about what lies behind the cosy facade of a BASIC interpreter or wonders why Pascal can't be like BASIC, I suggest this book as a good place to start.

Z80 Assembly Language Subroutines

By Lance A. Leventhal and Winthrop Saville
Published by Osborne/McGraw-Hill
$15.95, 498 pages, paperback
Reviewed by Paul J. Gans

Learning assembly language for a new computer is not easy for anyone. Even experienced programmers go through a period of adjustment when faced with a new machine. Novices are urged to study the architecture of the new machine to learn its op codes and addressing modes and to struggle as best they can to write *simple routines that have been written countless times before.*

More experienced programmers know that there is more to it than that. They do not become comfortable in the machine language of a new machine until they have mastered its idioms. *Z80 Assembly Language Subroutines* is a book of idioms for the Z80. It is a wonderful book that can be read with profit by anyone not yet able to speak idiomatic Z80.

The book begins with an explanation of Z80 architecture (with a section for experienced programmers, pointing out distinguishing features and difficulties with the Z80) and the Z80 instruction set. The instruction set is presented from a programmer's point of view. Thus instructions are grouped according to function. For instance, under "Storing Registers in Memory" follow direct storage of registers, indirect storage of registers, indexed storage of registers, etc. The instruction set is covered fully with many examples.

Chapter 2, entitled "Implementing Additional Instructions and Addressing Modes," is filled with instances of common operations requiring more than a single instruction. These operations occur repeatedly in real programming. The novice must, each time, figure out how best to perform them. These operations are, in fact, idioms, and are at the fingertips of experienced programmers. For example, how does one compare the contents of the HL register pair to a 16-bit immediate value? On page 81, Leventhal and Saville give two methods, one requiring two instructions, the other, three. Chapter 3 is devoted to common programming errors and includes such topics as using the flags incorrectly, confusing registers and register pairs, and handling arrays incorrectly.

The bulk of the book, however, is devoted to assembly language subroutines. There are more than 50 of them, covering topics such as code conversion (binary to BCD, ASCII decimal to binary, EBCDIC to ASCII, etc.), array manipulation, arithmetic (multiple-precision binary division, etc.), string manipulation (compare, concatenate, etc.), array operations, input/output, and interrupts. Each subroutine listing is elaborately commented, entry and exit conditions are clearly given, and execution times (in machine cycles) are provided. There is an introductory explanation of the function of each subroutine and a description of how it works. Best of all, test data and results are given. Much can be learned by examination of these subroutines, and, better yet, they are useful.

The Z80 instruction mnemonics used are Zilog's, which I believe are superior to the multiple and incompatible sets of pseudo-8080 mnemonics in common use. All notational conventions are explained. Appendices contain a detailed Z80 instruction set summary with reproductions of the Zilog instruction charts, a programming reference to the Z80 PIO, and an ASCII character set chart. The book concludes with a 23-page glossary and a useful index.

Mathematical Theories of Optimization; Proceedings, S. Margherita Ligure 1981; Lecture Notes in Mathematics Series No. 979

Edited by A. Dold and B. Eckmann
Published by Springer-Verlag
$14.50, 268 pages, paperback
Reviewed by Robert Irving

Optimization is the science of obtaining the maximum return for investment of resources under a given set of conditions. The field is also known as "operations research" from its initial application in World War II. The most common applications lie in the areas of inventory control, product selection, and manufacturing scheduling.

Engineers and computer scientists in the field of optimization are usually looking for applied solutions to real problems. They generally want either an algorithm or a program which will solve the problem at issue. Mathematicians, on the other hand, are not necessarily looking for solutions as such. They are more interested in the nature of the statement of the problem. Such things as "existence theorems" (does a solution exist?) and "robustness" (does the solution apply over a broad range of conditions?) are of more import than a particular solution.

This book is pure mathematics with potential practical application. Consequently, it is of little direct interest to most computer programmers.

Ultimately, some of the articles may lead to algorithms of great importance. For example, the article by L. Cesari on "Existence of Solutions and Existence of Optimal Solutions" may someday encourage other mathematicians to produce algorithms with which we can optimize some problems not now covered, but possible.

On the other hand, I see little to no potential use for a discussion by A. Bensoussan on page 71, entitled "On the Production Smoothing Problem," in which he extends a discrete case to a continuous case. First, I know of no "continuous" production process. Even oil refineries, which work with large volumes of material, operate on a "batch" basis. No two batches have the same characteristics, hence the run will be "continuous" only for the length of the batch. Second, all of us who make mathematical models on the computer know that we are dealing with discrete processes, since that is the only kind of process the computer can deal with.

Unless you are a "pure" mathematician interested in the rigors of that field, I suggest you forgo buying this volume.

Teletext and Videotex in the United States

By John Tydeman, Hubert Lipinski, Richard P. Adler, Michael Nyhan, and Laurence Zwimpfer
Published by McGraw-Hill
$30.00, 314 pages
Reviewed by Susan Bowers

Teletext (one-way information services) and Videotex (two-way information services) are new technologies created by the marriage of computing and communications. They emerged in England and France in the 1970's, have been growing rapidly in Europe, Canada, and Japan, and are currently becoming important tools in this country.

Teletext and Vidotex in the United States is a look at Teletex and Videotex from a futurist viewpoint. Using careful research and documentation, this in-depth overview deals with three classes of services: public data bases, closed user groups, and private systems. Full of charts and tables, the book looks at the present and future states of this technology in the United States. The first five chapters deal with defining the phenomenon, detailing its history, assessing the current state of information services, and discussing possible future applications. The rest of the book develops a model which studies the possible extent and impact of Teletext and Videotex in the United States by the end of this century.

The book emphasizes such public policy concerns as the effect of those technologies on society, legal aspects, security, privacy, standards, consumer protection, international ramifications, advertising liability, and impartiality. The modeling and forecasting are organized and carefully done, covering the possible futures from several perspectives. There is an impressive appendix on policy issues and an extensive bibliography, as well as a comprehensive index.

Teletext and Videotex in the United States provides an interesting look at future trends in the information services. Several parts of it would add much to a study of computers in society; it could also be an excellent textbook for a course on the future impact and direction of information services in the United States.

■■J

OF INTEREST

by Michael Wiesenberg

Stop Noise

Kalglo's **IPC** international model computer power control console with surge suppression and power line noise filter has six individually switched IEC-320/CEE-22 outlets in dual separately filtered banks of three each. It absorbs 318 joules at 18,200 amp pulse, is rated at 10 amps maximum with 100 or 260 VAC operation, 50 or 60 Hz, weighs 2.5 pounds, and costs $150. **Reader Service No. 101.**

The **Lemon AC Surge Protector** from Electronic Protection Devices gives protection in excess of 6000 volts and 200 amps, with instantaneous voltage clamping including grounding to prevent flashover to other ports and damage to equipment. The company thinks its product is so good that insurance by Lloyd's of London is included, protecting user hardware up to $2500 from damage caused by transient voltage surges. The Lemon lists for $59.95. **Reader Service No. 103.**

Buffer It

Big-Buffer from Mikrocomputertechnik of Germany is a Centronics-compatible hardware spooler for parallel printers. The 8K version is $170, the 32K version is $195, the 64K is $256, the 96K is $317, and the 120K is $363. You get all cables and connectors (36-pin Centronics with female input and male output). Power comes from the printer using pin 18 of the Centronics connector, or you can purchase a separate 110V, 60Hz power supply for $30. These prices are based on the exchange rate at the time of the release, 2.60 DM to $1 US, and are subject to change. **Reader Service No. 105.**

Can't Rip It

Micro Format has **Flexi Disk Envelopes** made of Dupont's Tyvek, a supposedly tear- and wear-proof substance that looks and feels like paper. They included a sample with their press release, and I couldn't tear the stuff. Each package has 25 envelopes, and you get five packages postpaid for $25. **Reader Service No. 107.**

TRS-80 Assembly Language Programming

TRS-80/Z80 Assembly Language Library by Craig Lindley from WG-Books has 75 program listings to thoroughly explain assembly language on the TRS-80, and includes two TRSDOS disks with 33 source and object files, for $34.97. **Reader Service No. 109.**

Dr. Dobb, Meet Dr. Logo

Dr. Logo, from Digital Research, is an advanced version of Logo for 16-bit micros, in particular the IBM PC and XT. It uses turtle graphics, just like the popular version of the language. Written in C, Dr. Logo is transferable to other systems. You get a larger work space, with over 10,000 nodes of memory, built-in help commands, a full-screen procedure editor (with on-line help as well). Also supported are comments, indentation, user-defined words, windowing, upper and lower case, new primitives, string processing, line editing, and debugging with a trace mode and a watch mode. The latter is for checking or modifying variables or expressions immediately after execution of a statement. Cross referencing, double-precision floating point, transcendental functions, and a $149.95 price tag are part of the package as well. **Reader Service No. 111.**

Go Forth on PC

HSFORTH, from Harvard Softworks, is Forth-79 for the PC and other 8086 and 8088 machines. Word names, definition lists, and machine codes use separate segments, permitting programs up to 192K. HSForth runs the Sieve of Eratosthenes in 47 seconds, as opposed to 70 to 140 on other Forths, and 2000 for interpreted BASIC. Auto-Opt and the Micro-Asm extension reduce the time to five seconds. Even optimized machine code, according to Harvard Softworks, runs no faster than five seconds. $220. **Reader Service No. 113.**

Things I've Tried

Usually the products I write about in this column are picked from a stack of press releases sent each month to *DDJ*. I rarely have the product in front of me when writing about it, and descriptions here are not intended to be endorsements.

Once in a while I do get the opportunity to try out products, and I'd like to share a few with you.

You may have gathered from previous columns that I like inexpensive computers, particularly the Timex products. One of the wonderful things you can do with a TS1000 or 1500 is learn Z80 assembly language. BASIC on a Timex runs achingly slow, particularly since some 60% of the microprocessor's time is spent updating the screen. Machine code programs, however, run literally hundreds of times as fast as BASIC ones. Several good books have been written about machine code programming on the TS1000. The best of these are two by Dr. Ian Logan, whose knowledge of the ZX80 and 81 (predecessors of the TS1000) is exceeded only by that of Clive Sinclair (inventor of these and the Spectrum, a version of which should now be available, modified by Timex and called the 2068).

You cannot write effective code for these little wonders without knowing the entry points for the various system variables. Melbourne House, an international publishing company, has Dr. Logan's *Understanding Your ZX81 ROM*, $14.95, and *The Complete TS1000/Sinclair ZX81 ROM Disassembly*, $19.95, and their own apparently authorless *Machine Language Programming Made Simple for Your Sinclair & Timex TS1000*, $14.95. I have all three of these, and they are excellent. Melbourne also offers *Not Only 30 Programs for the Sinclair ZX81*, $9.95, program listings that ingeniously all run in the "unexpanded" TS1000 (2K RAM), and what appears to be the ultimate hardware manual for the two "simpler" machines, *The Ins and Outs of the Timex TS1000 & ZX81*, by Don Thomasson, $12.95. Add $2 for p.&h., and residents of California, Maryland, and Tennessee add sales tax. Melbourne carries software and books for Timex, TRS80, Vic 20, and Commodore 64. They will send you a free catalog and

are looking for contributions from authors. **Reader Service No. 115.**

Far better than hand-assembling your machine code programs and then trying to POKE them into various memory locations is to use an assembler. International Publishing & Software Inc. offers **ZX Assembler.** It has its own editor. You write programs using standard Z80 mnemonics. No need to figure jump addresses; just use labels and the assembler figures them for you. Programs assemble in seconds and execute either directly from ZX Assembler's monitor mode or from a program that executes with a USR statement where the code is in a REM statement created by the assembler. The assembler also retains the source code in a second REM statement.

You can SAVE the entire program on tape, and alter, modify, and reassemble it. Those familiar with machine code on the TS will appreciate this method, because code loaded in other areas is either difficult to save or cannot use relative addressing. In the monitor mode you can move whole blocks of code from one memory location to another, examine and modify any location of RAM, inspect and modify registers, and search the entire memory for any byte sequence. ZX Assembler costs $14.95. International Publishing provides a free catalog and the name of the nearest dealer that handles this and other useful products, including **ZX Bug,** $14.95, for debugging, editing, and running machine codes, **Toolkit,** $14.95, to add nine powerful commands, including a renumber that changes all GOTO and GOSUB references, **Fastload,** $19.95, to load any program four to six times as fast, **ZX Forth,** $29.95, for the ease of BASIC with machine code speed, and a stack of clever games. **Reader Service No. 117.**

If I knew that a product did not live up to its claims, I would not list it here. Usually I have no way of knowing that, because I just go by the press releases, which, naturally, are never unfavorable. Such was the case with Memotech, whose various memory modules and auxiliary keyboard for the TS1000 I cited a few months back. I believed their hype myself and shortly afterward bought their keyboard and 32K memory for my system. That first 32K memory didn't work at all, and I took it back to the store. The replacement 32K refused to work in conjunction with either Memotech's keyboard or the Timex printer. This was not much use to me since I needed the printer for listings during program development. I took my entire system to the store, and *they* could get the 32K memory and auxiliary keyboard to work with their systems only separately, not together.

I called Memotech several times for help, and they kept insisting that they had tested a setup identical to mine with no problems. Finally they agreed to look at my system if I would send it to them and, if either of their products was defective, to replace or fix it. I called them when the system was in their hands, and the technician to whom I spoke said he had set up my system and it had been running perfectly for ten minutes, and he could find nothing wrong with it. But while I was on the phone with him he said, "Wait a minute. It's failing." He conceded that there "might" be a problem with the memory module, which Memotech replaced. They then sent back my system.

When the system returned, again I could make the 32K memory work by itself only, not together with either the Timex printer or Memotech's keyboard. The keyboard, too, only worked by itself. The store from which I bought the memory unit was kind enough to take it back in exchange for a Timex 16K memory (and even refunded the difference in price!), with which I have had no problems. I sent the keyboard back to Memotech, requesting a refund, citing the "free ten-day trial period" clause in their warranty, stating that, while the ten days had elapsed all the problems had begun the instant I began using the keyboard and that I had been in contact with them right from the start. They refused to give me a refund, claiming that the refund applied only to those who had purchased directly from them, but instead returned a replacement for my keyboard. If I wanted a refund, they told me, I would have to get it where I bought the keyboard.

The store, of course, did not want to give me a refund, because the 1500 would soon be released with its "real" keyboard, and auxiliary keyboards for Timex computers would soon be obsolete. They said that I should contact Memotech for a refund. (My guess is that Memotech is stuck with a warehouse full of these useless keyboards.) As I expected, the replacement worked no better than the original. At this point I no longer had their 32K memory, but the keyboard worked with neither Timex's printer nor the 16K memory.

By this time I had heard from the South Bay Area Timex/Sinclair Users' Group (mentioned in my October column) that Memotech was offering a free "modification" to purchasers of their 32K memories. It seems several other people had discovered that *their* memories did not work with Timex printers. Yet all the time I had been in communication with Memotech they insisted there was no incompatibility between their products and Timex printers; furthermore, they had never heard of anyone having any problems like mine. *Altogether, I went through three 32K memory modules and two keyboards without finding even one I could use.*

Now *Dr. Dobb's* has received a press pack announcing Memotech's $595 **MTX-512** World Class Computer™. It has a Z80A 4MHz processor with 80K standard RAM (expandable to 512K; 16K is dedicated to video). Among other things, it comes with Oxford BASIC and a language called NODDY, a real time clock, and a 79-key keyboard. I hope this product is better than the peripherals I couldn't use, and that others will encounter more competent service and better support and attitude than I did. I know *I'll* never get the chance to find out. **Reader Service No. 119.**

■■J

Index

16-Bit Software Toolbox (Column) 5, 58, 111, 214, 278, 328, 400, 551, 618, 707
6502, Instructions of 270
6800, Instructions of 270
68000
  Forth and the Motorola 68000 495
  Forth Assembler 502
6809
  Cross Assembler 272
  Threaded Code 145
8080
  to Z80 Translator 198
  fig-Forth Directory and File System 180
8088 Line Generator Subroutine 330

ACTxx Cross Assemblers 478
ADA Subset for Micros *see* AUGUSTA
AUGUSTA
  Part I: An ADA Subset for Micros 14
  Part II: P-Code Interpreter 114
  Part III: Recursive Descent Compilers 231
  Part IV: The AUGUSTA Compiler (Continued) 348
Algorithms
  Binary Magic Numbers 176
  Fast Circle Routine (IBM PC/IBM Pascal) 247
  Fast Divisibility 287
  Simple Graphics for Printer (Pascal) 692
Anatomy of a Digital Vector and Curve Generator 380
Apples, Fast Circles for 723
Assembler, 68000 Forth 502
Assembly Routine, Integer Square Route 216
Attaching a Winchester Hard Disk to the S-100 Bus 600

B-Tree ISAM Concepts 289
BASIC
  A Common-Sense Guide to Faster, Smaller BASIC 147
  Program Linkage, Forth to BASIC 64
Baden, W. 692
Barnhart, J. 495
Bartholomew, A. 289
BDOS
  Calls for C 293
  Function Calls 318
Bergman, E. E. 55
Binary Magic Numbers: Some Applications and Algorithms 176
BIOS Calls for C 293
Blum, R. 275, 334, 396, 470, 546, 627, 702
Bolstad, T. 293

C (*see also* Small C)
  C Screen Editor, Enhancing 251
  CP/M BDOS and BIOS Calls for 293
  RED: A Better C Screen Editor 361, 446
  /UNIX Programmer's Notebook 567, 720
Comments on "Fifth Generation Computers" 151
Common-Sense Guide to Faster, Smaller BASIC 147
Compilers
  AUGUSTA: Recursive Descent Compilers 231
  Design 149
Computers, "Fifth Generation" 151, 411
Coroutines, Program Linkage, Forth to BASIC 64
Cortesi, D.E. 30, 90, 212, 285, 392, 415, 492, 565, 639, 719
CP/M
  Application Note Fourteen 703
  CP/M Exchange (Column) 46, 72, 152, 167, 275, 334, 396, 470, 546, 627, 702
  Linking CP/M Functions to Your High-Level Program 70
  UNIX to CP/M Floppy Disk File Conversion 570
  Utilities
    BDOS and BIOS Calls for C 293
    DIR.ASM 72
    RECLAIM 170
CP/M-80 Expansion Card for the Victor 9000 92
CP/M Plus
  BIOS 546
  Direct Access Disk I/O Benchmarks 415
Cross Assembler, ACTxx 478
Curve Generator and Digital Vector, Anatomy of 380

Daetwyler, D. 298
Dates, Julian 323
DDA Vector Generator 380
Deppe, C. 298
Dettmann, T.R. 272
Digital Vector and Curve Generator, Anatomy of 380
Disassembler, REVAS 95
DML Parser 662
Dobb's Ex Machina 57
Doherty, M.J. 151
Dr. Dobb's Clinic (Column) 90, 212, 285, 392, 415, 492, 565, 639, 719
Dumb Terminals, Cursor Control 733
Duncan, R. 4, 58, 111, 214, 278, 328, 400, 551, 618

Editorials 4, 52, 165, 227, 345, 413, 489, 564, 638, 717
Enhancing the C Screen Editor 251
Enright, M.T. 725
Epson, Using the Bit Plot Graphics 608

Fast Divisibility Algorithms 287
Fast Matrix Operations in Forth 383, 433
Faster Circles for Apples 723
Favitta, M. 218
Fifth Generation Computers 151, 411
Forth
  68000 Forth Assembler 502
  8080 fig-Forth Directory and File System 184
  and the Motorola 68000 495
  Coding Standards 538
  Fast Matrix Operations 315, 383, 433
  Game of GO 516
  Nondeterministic Control Words 510
  Precompiled Modules 529
  Program Linkage to BASIC 64
  Say" Forth Votrax Driver" 194
  Series Expansion 311
  Sort 542
  Syntax 668
Forth-83 and Vocabularies 535
A Fundamental Mistake in Compiler Design 149
Freed, E.E. 176

The Game of Life on the IBM PC 304
Garfinkel, S.L. 304
Gates, W.C. 311
Gay, D. 608
Glass, H. 688
Goldman, O. 600
Goodman, R. 268
Gordon, H.T. 145, 287
Graphics
  Faster Circles for Apples 723
  Printing Graphics Using the IBM PC 298
  Simple Graphics for Printer 692
  Using the Epson Bit Plot Graphics 608
Gregory, R. 21
Grigonis, R. 441

Harrison, H. 4, 24, 57, 92
Hardware Reviews *see* Reviews, Hardware
Harry, D. 662

Head, G. 46, 72, 152, 167
Howard, A.D. 251
Hunt, D.S. 29

IBM PC
  The Game of Life on 304
  Printing Graphics with 298
  Sizing Memory on 401
Integer
  Division 533
  Square Root 216
Irving, R. 147
ISAM, B-Tree Concepts 289

Jolly, G.W. 273, 480
Jones, D-W. 318
JRT Pascal: Another Look 29
Julian Dates for Microcomputers 323

Kernel for the MC 68000 646
King, G. 323

Lee, D.L. 247
Linking CP/M Functions to Your High-Level Program 70
Lyons, V. 97

Magic Wand Utility 393
Manning, M. 542
Matrix Operations in Forth 315, 383, 433
MC 68000, Kernel for 646
McCabe, C.K. 194
McKeon, B. 126
McWORDER: A Tiny Text Editor 428
McWorter, W.A. 428, 733
Mitchell, E. 14, 114, 348
Monroe, A.J. 184
Moran, J. 700
Morgenstern, L. 64
Motorola 68000, Forth and 495
Murphy, W.V. 170

Newberry, S. 478
Nondeterministic Control Words in Forth 510

Odette, L.L. 510
Of Interest (Column) 47, 99, 159, 221, 280, 336, 406, 483, 557, 631, 712

P-Code Interpreter 114
Parser, DML 662
Pascal, JRT 29
Passe, S. 646
Perry, M.A. 502
PISTOL: A Forth-Like Portably Implemented STack Oriented Language 55
Precompiled Forth Modules 529
Printer, Simple Graphics for 692
Program Linkage by Coroutines: Forth to BASIC 64
Pulier, M.L. 723

Radio Shack TRS-80 *see* TRS-80
Ream, E.K. 361, 446
RECLAIM: A File Reclamation Utility for Destroyed Directories 170
Recursive Descent Compilers 231
RED: A Better C Screen Editor 361, 446
REVAS Disassembler 95
Reviews, Book 96, 273, 480, 700
Review, Hardware
  CP/M-80 Expansion Card for the Victor 9000 92
Reviews, Software
  ACTxx Cross Assemblers 478
  JRT Pascal 29
  REVAS Disassembler 95
  Unica and XM-80 218
  ZAS Z8000 Software Development Package 272
Rothstein, J.B. 96, 700
Ruzinsky, S.A. 315, 383, 433

S-100 Bus, Attaching a Winchester to 600
Say" Forth Votrax Driver" 194
SBC, TSX and TXS: Instructions of the 6800 and 6502 270
Scarpelli, A.T. 198
Screen Editors
  Enhancements 251
  RED: A Better C Screen Editor 361, 446
SEND and RCV, a Forth Implementation of the XMODEM Protocol 523
Serial to Parallel: A Flexible Utility Box 422
Series Expansion in Forth 311
Shaw, G.W. II 535
Shiffman, R. 380
Shifts and Rotations on the Z80 268
Signed Integer Division 533
Skjellum, A. 567, 720
Small-C
  Compiler, V.2 31
  Help Facility 583
  Operating System 126
Smith, R.L. 533
Software Reviews *see* Reviews, Software
Solntseff, N. 529
Some Forth Coding Standards 538
Staneff, J. 583
Stetson, E.B. 273
Stone Age Computers: 6,000 Years of Computing Science 36

TRS-80 8080 to Z80 Translator 198
Tan, B.T.G. 270
Taylor, D. 216
Taylor, R. 523
Text Editor, McWORDER 428
Towards a More Writeable Forth Syntax 688
Trace Function, DDT 318
Translator, TRS-80 8080 to Z80 198

Unica 218
UNIX
  C/UNIX Programmer's Notebook 567, 720
  to CP/M Floppy Disk File Conversion 570
Using the Epson Bit Plot Graphics 608

Victor 9000, CP/M Expansion Card 92

Watson, T. 98
Weiger, W. 70
When a Page Is Not a Page: Forth-83 and Vocabularies 535
Wiesenberg, M. 44, 99, 159, 221, 280, 336, 406, 483, 557, 712
Wiggins, R. 53, 165, 227, 345, 413, 489, 564, 638, 717
Wilcox, A.D. 422
Willoughby, S. 95
Wilson, W.E. 570
Winchester Hard Disk, Attaching to the S-100 Bus 600
Wischmeyer, E. 538

XM-80 and Unica 218
XANADU: Hypertext from the Future 21
XMODEM Protocol, Forth Implementation 523

Yes, You Can Trace Through BDOS 318

Z80
  8080 to Z80 Translator 198
  Shifts and Rotations 268
ZAS Z8000 Software Development Package 272

The Dr. Dobb's Series

"There's more creativity, wild ideas, and raw enthusiasm here per page than you are likely to find anywhere else in the personal computing press."
Creative Computing

Volume 1

Chronicles the advent of the microcomputer era in 1976. Always pertinent for bit crunching and byte saving, home-brew computer projects, and the technical history of home computing. Topics include: Tiny BASIC, the first words on CP/M, Speech Synthesis, Floating Point Routines, 6502 Disassembler for the Apple, and much more.

Volume 2

The small computer emerges as a powerful tool for the modern age in these issues from 1977. Topics include: Lawrence Livermore Lab's BASIC, Dr. Starkweather's PILOT, Using a Modem, String Handling Techniques, Ciphers, Turtle Graphics, micro utilities.

Volume 3

Explores the foundations of the mass market computer industry in 1978. Topics include: Programming in BASIC, PILOT, and Pascal, RAM Memory Testers, Apple utility programs, S-100 Bus Standard, STRUBAL: A Structured BASIC Compiler.

Volume 4

Heralds the widespread acceptance of the microcomputer in America. Innovative ideas and articles guide the reader through the age of discovery in 1979. Topics include: Selecting Business Software, Microcomputer Speech and Music, Information Networks, interfacing techniques.

Volume 5

Focuses on the technological promise of the modern microcomputer and the creative challenge facing programmers in 1980. Topics include: The revolutionary impact of CP/M, C programming and the UNIX operating system, survey of computer networks, software portability, introduction to FORTH, compiler writing.

Volume 6

The microcomputer enters the mainstream—these issues assess the revolutionary impact of microcomputers on the individual and on our society in 1981. Topics include: Computer Conferencing, The Power of FORTH, 8- and 16-Bit Technology, Rubik's Cube Simulator, An Adventure Game Development System.

Volume 7

Examines the potential of powerful 16-bit micros including the birth of the IBM PC in 1982. Topics include: In-depth coverage of FORTH, 68000, 8088 programming, C software tools, utilities for CP/M including a spelling checker, using bulletin boards, and more.

Volume 8

DDJ turns pro. Some of the most powerful, professional programmer's tools ever published in a magazine are in this volume. Jim Hendrix's Small C compiler. Ed Ream's RED screen editor. A microcomputer subset of the Defense Department's official programming language, Ada. C and Forth and 68000 software. Because the magazine increased in size in 1983, this volume is bigger and better than ever.

DR. DOBB'S BOOKS
M&T Publishing, Inc.

ISBN 0-934375-00-3